A Query Language for Formal Mathematical Libraries

Florian Rabe

Jacobs University Bremen, Germany

Abstract. One of the most promising applications of mathematical knowledge management is search: Even if we restrict attention to the tiny fragment of mathematics that has been formalized, the amount exceeds the comprehension of an individual human.

Based on the generic representation language MMT, we introduce the mathematical query language QMT: It combines simplicity, expressivity, and scalability while avoiding a commitment to a particular logical formalism. QMT can integrate various search paradigms such as unification, semantic web, or XQuery style queries, and QMT queries can span different mathematical libraries.

We have implemented QMT as a part of the MMT API. This combination provides a scalable indexing and query engine that can be readily applied to any library of mathematical knowledge. While our focus here is on libraries that are available in a content markup language, QMT naturally extends to presentation and narration markup languages.

1 Introduction and Related Work

Mathematical knowledge management applications are particularly strong at large scales, where automation can be significantly superior to human intuition. This makes search and retrieval pivotal MKM applications: The more the amount of mathematical knowledge grows, the harder it becomes for users to find relevant information. Indeed, even expert users of individual libraries can have difficulties reusing an existing development because they are not aware of it. Therefore, this question has received much attention.

Object query languages augment standard text search with phrase queries that match mathematical operators and with wild cards that match arbitrary mathematical expressions. Abstractly, an object query engine is based on an index, which is a set of pairs (l, o) where o is an object and l is the location of o. The index is built from a collection of mathematical documents, and the result of an object query is a subset of the index. The object model is usually based on presentation MathML and/or content MathML/OpenMath [W3C03,BCC$^+$04], but importers can be used to index other formats such as LaTeX. Examples for object query languages and engines are given in [MY03,MM06,MG08,KŞ06,SL11]. A partial overview can be found in [SL11]. A central question is the use of wild cards. An example language with complex wild cards is given in [AY08]. Most generally, [KŞ06] uses unification queries that return all objects that can be unified with the query.

Property query languages are similar to object query languages except that both the index and the query use relational information that abstracts from the mathematical objects. For example, the relational index might store the toplevel symbol of every object or the "used-in" relation between statements. This approximates an object index, and many property queries are special cases of object queries. But property queries are simpler and more efficient, and they still cover many important examples. Such languages are given in [GC03,AS04] and [BR03] based on the Coq and Mizar libraries, respectively.

Compositional query languages focus on a complex language of query expressions that are evaluated compositionally. The atomic queries are provided by the elements of the queried library. SQL [ANS03] uses n-ary relations between elements, and query expressions use the algebra of relations. The SPARQL [W3C08] data model is RDF, and queries focus on unary and binary predicates on a set of URIs of statements. This could serve as the basis for mathematics on the semantic web. Both data models match bibliographical meta-data and property-based indices and could also be applied to the results of object queries (seen as sets of pairs); but they are not well-suited for expressions. The XQuery [W3C07] data model is XML, and query expressions are centered around operations on lists of XML nodes. This is well-suited for XML-based markup languages for mathematical documents and expressions and was applied to OMDoc [Koh06] in [ZK09]. In [KRZ10], the latter was combined with property queries. Very recently [ADL12] gave a compositional query language for hiproof proof trees that integrates concepts from both object and property queries.

A number of **individual libraries** of mathematics provide custom query functionality. Object query languages are used, for example, in [LM06] for Activemath or in Wolfram|Alpha. Most interactive proof assistants permit some object or property queries, primarily to search for theorems that are applicable to a certain goal, e.g., Isabelle, Coq, and Matita. [Urb06] is notable for using automated reasoning to prepare an index of all Mizar theorems.

It is often desirable to combine several of the above formalisms in the same query. Therefore, we have designed the query language QMT with the goal of permitting as many different query paradigms as possible. QMT uses a simple kernel syntax in which many advanced query paradigms can be defined. This permits giving a formal syntax, a formal semantics, and a scalable implementation, all of which are presented in this paper.

QMT is grounded in the MMT language (Module System for Mathematical Theories) [RK11], a scalable, modular representation language for mathematical knowledge. It is designed as a scalable trade-off between (i) a logical framework with formal syntax and semantics and (ii) an MKM framework that does not commit to any particular formal system. Thus, MMT permits both adequate representations of virtually any formal system as well as the implementation of generic MKM services. We implement QMT on top of our MMT system, which provides a flexible and scalable query API and query server.

Our design has two pivotal strengths. Firstly, QMT can be applied to the libraries of any formal system that is represented as MMT. Queries can even span

Declaration	Intended Semantics
base type a	a set of objects
concept symbol c	a subset of a base type
relation symbol r	a relation between two base types
function symbol f	a sorted first-order function
predicate symbol p	a sorted first-order predicate

Kind of Expression	Intended Semantics
Type T	a set
Query $Q : T$	an element of T
element query $Q : t$	an element of t
set query $Q : set(t)$	a subset of t
Relation $R < a, a'$	a relation between a and a'
Proposition F	a boolean truth value

Fig. 1. QMT Notions and their Intuitions

libraries of different systems. Secondly, QMT queries can make use of other MMT services. For example, queries can access the inferred type and the presentation of a found expression, which are computed dynamically.

We split the definition of QMT into two parts. Firstly, Sect. 2 defines QMT signatures in general and then the syntax and semantics of QMT for an arbitrary signature. Secondly, Sect. 3 describes a specific QMT signature that we use for MMT libraries. Our implementation, which is based on that signature, is presented in Sect. 4.

2 The QMT Query Language

2.1 Syntax

Our syntax arises by combining features of sorted first-order logic – which leads to very intuitive expressions – and of description logics – which leads to efficient evaluations. Therefore, our **signatures** Σ contain five kinds of declarations as given in Fig. 1.

For a given signature, we define four kinds of **expressions**: types T, relations R, propositions F, and typed queries Q as listed in Fig. 1. The grammar for signatures and expressions is given in Fig. 2.

The intuitions for most expression formation operators can be guessed easily from their notations. In the following we will discuss each in more detail.

Regarding **types** T, we use product types and power type. However, we go out of our way to avoid arbitrary nestings of type constructors. Every type is either a product $t = a_1 \times \ldots \times a_n$ of base types a_i or the power type $set(t)$ of such a type. Thus, we are able to use the two most important type formation operators in the context of querying: product types arise when a query contains multiple query variables, and power types arise when a query returns multiple

Signatures	$\Sigma ::= \cdot \mid \Sigma,\ a : \mathit{type} \mid \Sigma,\ c < a \mid \Sigma,\ r < a, a$
	$\mid \Sigma,\ f : T, \ldots, T \to T \mid \Sigma,\ p : T, \ldots, T \to \mathit{prop}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : T$
Simple Types	$t ::= a \times \ldots \times a$
General Types	$T ::= t \mid \mathit{set}(t)$
Relations	$R ::= r \mid R^{-1} \mid R^{*} \mid R; R \mid R \cup R \mid R \cap R \mid R \setminus R$
Propositions	$F ::= p(Q, \ldots, Q) \mid \neg F \mid F \wedge F \mid \forall x \in Q.F(x)$
Queries	$Q ::= c \mid x \mid f(Q, \ldots, Q) \mid Q * \ldots * Q \mid Q_i$
	$\mid R(Q) \mid \bigcup_{x \in Q} Q(x) \mid \{x \in Q \vert F(x)\}$

Fig. 2. The Grammar for Query Expressions

results. But at the same time, the type system remains very simple and can be treated as essentially first-order.

Regarding **relations**, we provide the common operations from the calculus of binary relations: dual/inverse R^{-1}, transitive closure R^*, composition $R; R'$, union $R \cup R'$, intersection $R \cap R'$, and difference $R \setminus R'$. Notably absent is the absolute complement operation R^{C}; it is omitted because its semantics can not be computed efficiently in general. Note that the operation R^{-1} is only necessary for atomic R: For all other cases, we can put $(R^*)^{-1} = (R^{-1})^*$, $(R; R')^{-1} = R'^{-1}; R^{-1}$, and $(R * R')^{-1} = R^{-1} * R'^{-1}$ for $* \in \{\cup, \cap, \setminus\}$.

Regarding **propositions**, we use the usual constructors of classical first-order logic: predicates, negation, conjunction, and universal quantification. As usual, the other constructors are definable. However, there is one specialty: The quantification $\forall x \in Q.F(x)$ does not quantify over a type t; instead, it is relativized by a query result $Q : \mathit{set}(t)$. This specialty is meant to support efficient evaluation: The extension of a base type is usually much larger than that of a query, and it may not be efficiently computable or not even finite.

Regarding **queries**, our language combines intuitions from description and first-order logic with an intuitive mathematical notation. Constants c, variables x, and function application are as usual for sorted first-order logic. $Q^1 * \ldots * Q^n$ for $n \in \mathbb{N}$ and Q_i for $i = 1, \ldots, n$ denote tupling and projection. $R(Q)$ represents the image of the object given by Q under the relation given by R. $\bigcup_{x \in Q} Q'(x)$ denotes the union of the family of queries $Q'(x)$ where x runs over all objects in the result of Q. Finally, $\{x \in Q | F(x)\}$ denotes comprehension on queries, i.e., the objects in Q that satisfy F. Just like for the universal quantification, all bound variables are relativized to a query result to support efficient evaluation.

Remark 1. While we do not present a systematic analysis of the efficiency of QMT, we point out that we designed the syntax of QMT with the goal of supporting efficient evaluation. In particular, this motivated our distinction between the ontology part, i.e., concept and relation symbols, and the first-order part, i.e., the function and predicate symbols.

Indeed, every concept $c < t$ can be regarded as a function symbol $c : \mathit{set}(t)$, and every relation $r < a, a'$ as a predicate symbol $r : a, a' \to \mathit{prop}$. Thus, the ontology symbols may appear redundant — their purpose is to permit efficient evaluations. This is most apparent for relations. For a predicate symbol p :

$$\frac{n \text{ not declared in } \Sigma}{n \notin \Sigma} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \quad a \notin \Sigma}{\vdash \Sigma,\, a : \mathit{type}}$$

$$\frac{\vdash \Sigma \quad c \notin \Sigma \quad a : \mathit{type} \text{ in } \Sigma}{\vdash \Sigma,\, c < a} \qquad \frac{\vdash \Sigma \quad r \notin \Sigma \quad \left(a_i : \mathit{type} \text{ in } \Sigma\right)_{i=1}^{2}}{\vdash \Sigma,\, r < a_1, a_2}$$

$$\frac{\vdash \Sigma \quad f \notin \Sigma \quad \left(\vdash_\Sigma T_i : \mathit{type}\right)_{i=1}^{n+1}}{\vdash \Sigma,\, f : T_1, \ldots, T_n \to T_{n+1}} \qquad \frac{\vdash \Sigma \quad p \notin \Sigma \quad \left(\vdash_\Sigma T_i : \mathit{type}\right)_{i=1}^{n}}{\vdash \Sigma,\, p : T_1, \ldots, T_n \to \mathit{prop}}$$

Fig. 3. Well-Formed Signatures

$a, a' \to \mathit{prop}$, evaluation requires a method that maps from $[\![a]\!] \times [\![a']\!]$ to booleans. But for a relation symbol $r < a, a'$, evaluation requires a method that returns for any u all v such that $(u, v) \in [\![r]\!]$ or all v such that $(v, u) \in [\![r]\!]$. A corresponding property applies to concepts.

Therefore, efficient implementations of QMT should maintain indices for them that are computed a priori: hash sets for the concept symbols and hash tables for the relation symbols. (Note that using hash tables for all relation symbols permits fast evaluation of all relation expressions R, which is crucial for the evaluation of queries $R(Q)$.) The implementation of function and predicate symbols, on the other hand, only requires plain functions that are called when evaluating a query.

Thus, it is a design decision whether a certain feature is realized by an ontology or by a first-order symbol. By separating the ontology and the first-order part, we permit simple indices for the former and retain flexible extensibility for the latter (see also Rem. 2).

Based on these intuitions, it is straightforward to define the **well-formed expressions**, i.e., the expressions that will have a denotational semantics. More formally, we use the **judgments** given in Fig. 5 to define the well-formed expressions over a signature Σ and a context Γ. The **rules** for these judgments are given in Fig. 3 and 4.

Judgment	Intuition
$\vdash \Sigma$	well-formed signature Σ
$\vdash_\Sigma T : \mathit{type}$	well-formed type T
$\Gamma \vdash_\Sigma Q : T$	well-typed query Q of type T
$\Gamma \vdash_\Sigma Q : T$	well-typed query Q of type T
$\vdash_\Sigma R < a, a'$	well-typed relation R between a and a'
$\Gamma \vdash_\Sigma F : \mathit{prop}$	well-formed proposition F

Fig. 5. Judgments

In order to give some meaningful examples, we will already make use of the symbols from the MMT signature, which we will introduce in Sect. 3.

$$\frac{\left(a_i : \mathit{type} \text{ in } \Sigma\right)_{i=1}^{n}}{\vdash_\Sigma a_1 \times \ldots \times a_n : \mathit{type}} \qquad \frac{\left(a_i : \mathit{type} \text{ in } \Sigma\right)_{i=1}^{n}}{\vdash_\Sigma \mathit{set}(a_1 \times \ldots \times a_n) : \mathit{type}}$$

$$\frac{c < t \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : \mathit{set}(t)} \qquad \frac{f : T_1, \ldots .T_n \to T \text{ in } \Sigma \quad \Gamma \vdash_\Sigma Q_i : T_i}{\Gamma \vdash_\Sigma f(Q_1, \ldots, Q_n) : T} \qquad \frac{x : T \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : T}$$

$$\frac{\Gamma \vdash_\Sigma Q_i : t_i \text{ for } i = 1, \ldots, n}{\Gamma \vdash_\Sigma Q_1 * \ldots * Q_n : t_1 \times \ldots \times t_n} \qquad \frac{\Gamma \vdash_\Sigma Q : t_1 \times \ldots \times t_n \quad i \in \{1, \ldots, n\}}{\Gamma \vdash_\Sigma Q_i : t_i}$$

$$\frac{\Gamma \vdash_\Sigma Q : \mathit{set}(t) \quad \Gamma, x : t \vdash_\Sigma Q'(x) : \mathit{set}(t')}{\Gamma \vdash_\Sigma \bigcup_{x \in Q} Q'(x) : \mathit{set}(t')}$$

$$\frac{\Gamma \vdash_\Sigma Q : t \quad \vdash_\Sigma R < t, t'}{\Gamma \vdash_\Sigma R(Q) : \mathit{set}(t')} \qquad \frac{\Gamma \vdash_\Sigma Q : \mathit{set}(t) \quad \Gamma, x : t \vdash_\Sigma F(x) : \mathit{prop}}{\Gamma \vdash_\Sigma \{x \in Q | F(x)\} : \mathit{set}(t)}$$

$$\frac{r < a, a' \text{ in } \Sigma}{\vdash_\Sigma r < a, a'} \qquad \frac{\vdash_\Sigma R < a, a'}{\vdash_\Sigma R^{-1} < a', a} \qquad \frac{\vdash_\Sigma R < a, a}{\vdash_\Sigma R^* < a, a}$$

$$\frac{\vdash_\Sigma R < a, a' \quad \vdash_\Sigma R' < a', a''}{\vdash_\Sigma R; R' < a, a''} \qquad \frac{\vdash_\Sigma R < a, a' \quad \vdash_\Sigma R' < a, a' \quad * \in \{\cup, \cap, \setminus\}}{\vdash_\Sigma R * R' < a, a'}$$

$$\frac{p : T_1, \ldots, T_n \to \mathit{prop} \text{ in } \Sigma \quad \Gamma \vdash_\Sigma Q_i : T_i}{\Gamma \vdash_\Sigma p(Q_1, \ldots, Q_n) : \mathit{prop}}$$

$$\frac{\Gamma \vdash_\Sigma F : \mathit{prop}}{\Gamma \vdash_\Sigma \neg F : \mathit{prop}} \qquad \frac{\Gamma \vdash_\Sigma F : \mathit{prop} \quad \Gamma \vdash_\Sigma F' : \mathit{prop}}{\Gamma \vdash_\Sigma F \wedge F' : \mathit{prop}}$$

$$\frac{\Gamma \vdash_\Sigma Q : \mathit{set}(t) \quad \Gamma, x : t \vdash_\Sigma F(x) : \mathit{prop}}{\Gamma \vdash_\Sigma \forall x \in Q.F(x) : \mathit{prop}}$$

Fig. 4. Well-Formed Expressions

Example 1. Consider a base type $\mathit{id} : \mathit{type}$ of MMT identifiers in some fixed MMT library. Moreover, consider a concept symbol $\mathit{theory} < \mathit{id}$ giving the identifiers of all theories, and a relation symbol $\mathit{includes} < \mathit{id}, \mathit{id}$ that gives the relation "theory A directly includes theory B".

Then the query *theory* of type $set(id)$ yields the set of all theories. Given a theory u, the query $includes^{*-1}(u)$ of type $set(id)$ yields the set of all theories that transitively include u.

Example 2 (Continued). Additionally, consider a concept $constant < id$ of identifiers of MMT constants, relation symbol $declares < id, id$ that relates every theory to the constants declared in it, a base type $obj : type$ of OpenMath objects, a function symbol $type : id \to obj$ that maps each MMT constant to its type, and a predicate symbol $occurs : id, obj \to prop$ that determines whether an identifier occurs in an object.

Then the following query of type $set(id)$ retrieves all constants that are included into the theory u and whose type uses the identifier v:

$$\{x \in (includes^*; declares)(u) \mid occurs(v, type(x))\}$$

2.2 Semantics

A Σ-model assigns to every symbol s in Σ a denotation. The formal definition is given in Def. 1. Relative to a fixed model M (which we suppress in the notation), each well-formed expression has a well-defined denotational semantics, given by the interpretation function $[\![-]\!]$. The semantics of propositions and queries in context Γ is relative to an assignment α, which assigns values to all variables in Γ. An overview is given in Fig. 6. The formal definition is given in Def. 2.

Judgment	Semantics
$\vdash_\Sigma T : type$	$[\![T]\!] \in \mathcal{SET}$
$\Gamma \vdash_\Sigma Q : t$	$[\![Q]\!]^\alpha \in [\![t]\!]$
$\Gamma \vdash_\Sigma Q : set(t)$	$[\![Q]\!]^\alpha \subseteq [\![t]\!]$
$\vdash_\Sigma R < a, a'$	$[\![R]\!] \subseteq [\![a]\!] \times [\![a']\!]$
$\Gamma \vdash_\Sigma F : prop$	$[\![F]\!]^\alpha \in \{0, 1\}$

Fig. 6. Semantics of Judgments

Definition 1 (Models). *A Σ-model M assigns to every Σ-symbol s a denotation s^M such that*

- a^M *is a set for* $a : type$
- $c^M \subseteq [\![a]\!]$ *for* $c < a$
- $r^M \subseteq [\![a]\!] \times [\![a']\!]$ *for* $r < a, a'$
- $f^M : [\![T_1]\!] \times \ldots \times [\![T_n]\!] \to [\![T]\!]$ *for* $f : T_1, \ldots, T_n \to T$
- $p^M : [\![T_1]\!] \times \ldots \times [\![T_n]\!] \to \{0, 1\}$ *for* $p : T_1, \ldots, T_n \to prop$

Definition 2 (Semantics). *Given a Σ-model M, the interpretation function $[\![-]\!]$ is defined as follows.*
Semantics of types:

- $[\![a_1 \times \ldots \times a_n]\!]$ *is the cartesian product* $a_1^M \times \ldots \times a_n^M$
- $[\![set(t)]\!]$ *is the power set of* $[\![t]\!]$

Semantics of relations:

- $[\![r]\!] = r^M$
- $[\![R^{-1}]\!]$ *is the dual/inverse relation of* $[\![R]\!]$*, i.e., the set* $\{(u, v) \mid (v, u) \in [\![R]\!]\}$

- R^* *is the transitive closure of* $[\![R]\!]$
- $R;R'$ *is the composition of* $[\![R]\!]$ *and* $[\![R']\!]$,
  i.e., the set $\{(u,w) |\text{ exists } v \text{ such that } (u,v) \in [\![R]\!],\ (v,w) \in [\![R']\!]\}$
- $R \cup R'$, $R \cap R'$, *and* $R \setminus R'$ *are interpreted in the obvious way using the union, intersection, and difference of sets*

Semantics of propositions under an assignment α*:*

- $[\![p(Q_1,\ldots,Q_n)]\!]^\alpha = p^M([\![Q_1]\!]^\alpha,\ldots,[\![Q_n]\!]^\alpha)$
- $[\![\neg F]\!]^\alpha = 1 \quad \text{iff} \quad [\![F]\!]^\alpha = 0$
- $[\![F \wedge F']\!]^\alpha = 1 \quad \text{iff} \quad [\![F]\!]^\alpha = 1$ *and* $[\![F']\!]^\alpha = 1$
- $[\![\forall x \in Q.F(x)]\!]^\alpha = 1 \quad \text{iff} \quad [\![F(x)]\!]^{\alpha,x/u} = 1 \quad \text{for all } u \in [\![Q]\!]^\alpha$

Semantics of queries $\Gamma \vdash_\Sigma Q : T$ *under an assignment* α*:*

- $[\![c]\!]^\alpha = c^M$
- $[\![x]\!]^\alpha = \alpha(x)$
- $[\![f(Q_1,\ldots,Q_n)]\!]^\alpha = f^M([\![Q_1]\!]^\alpha,\ldots,[\![Q_n]\!]^\alpha)$
- $[\![R(Q)]\!]^\alpha = \{u \in [\![a']\!] \,|\, ([\![Q]\!]^\alpha, u) \in [\![R]\!]\}$ *for a relation* $\vdash_\Sigma R < a, a'$ *and a query* $\Gamma \vdash_\Sigma Q : a$
  informally, $[\![R(Q)]\!]^\alpha$ *is the image of* $[\![Q]\!]^\alpha$ *under* $[\![R]\!]$
- $[\![\bigcup_{x \in Q} Q'(x)]\!]^\alpha$ *is the union of all sets* $[\![Q'(x)]\!]^{\alpha,x/u}$ *where* u *runs over all elements of* $[\![Q]\!]^\alpha$
- $[\![\{x \in Q | F(x)\}]\!]^\alpha$ *is the subset of* $[\![Q]\!]^\alpha$ *containing all elements* u *for which* $[\![F(x)]\!]^{\alpha,x/u} = 1$

Remark 2. It is easy to prove that if all concept and relation symbols are interpreted as finite sets and if all function symbols with result type $set(t)$ always return finite sets, then all well-formed queries of type $set(t)$ denote a *finite* subset of $[\![t]\!]$. Moreover, if the interpretations of the function and predicate symbols are computable functions, then the interpretation of queries is computable as well. This holds even if base types are interpreted as infinite sets.

2.3 Predefined Symbols

Symbol	Type	Semantics
$\{\_\}$	$: t \to set(t)$	the singleton set
$\_ \doteq \_$	$: t, t \to prop$	equality
$\_ \in \_$	$: t, set(t) \to prop$	elementhood

Fig. 7. Predefined Symbols

We use a number of predefined function and predicate symbols as given in Fig. 7. These are assumed to be implicitly declared in every signature, and their semantics is fixed. All of these symbols are overloaded for all simple types t. Moreover, we use special notations for them.

All of this is completely analogous to the usual treatment of equality as a predefined predicate symbol in first-order logic. The only difference is that our slightly richer type system calls for a few additional predefined symbols.

It is easy to add further predefined symbols, in particular equality of sets (which, however, may be inefficient to decide) and binary union of queries. We omit these here for simplicity.

2.4 Definable Queries

Using the predefined symbols, we can define a number of further useful query formation operators:

Example 3. Using the singleton symbol $\{\_\}$, we can define for $\Gamma \vdash_\Sigma Q : set(t)$ and $\Gamma, x : t \vdash_\Sigma q(x) : t'$

$$\{q(x) \,:\, x \in Q\} \;:=\; \bigcup_{x \in Q} \{q(x)\} \qquad \text{of type } set(t').$$

It is easy to show that, semantically, this is the replacement operator, i.e., $[\![\{q(x) \,:\, x \in Q\}]\!]^\alpha$ is the set containing exactly the elements $[\![q(x)]\!]^{\alpha,x/u}$ for any $u \in [\![Q]\!]^\alpha$.

Example 4 (SQL-style Queries). For a query $\vdash_\Sigma Q : set(a_1 \times \ldots \times a_N)$, natural numbers $n_1, \ldots, n_k \in \{1, \ldots, N\}$, and a proposition $x_1 : a_1, \ldots, x_N : a_N \vdash_\Sigma F(x_1, \ldots, x_n) : prop$, we write

$$\textbf{select } n_1, \ldots, n_k \textbf{ from } Q \textbf{ where } F(1, \ldots, N)$$

for the query

$$\{x_{n_1} * \ldots * x_{n_k} \;:\; x \in \{y \in Q \,|\, F(y_1, \ldots, y_N)\}\}$$

of type $set(a_{n_1} \times \ldots \times a_{n_k})$.

Example 5 (XQuery-style Queries). For queries $\vdash_\Sigma Q : set(a)$ and $x : a \vdash_\Sigma q'(x) : a'$ and $x : a, y : a' \vdash_\Sigma Q''(x, y) : set(a'')$, and a proposition $x : a, y : a' \vdash_\Sigma F(x, y) : prop$, we write

$$\textbf{for } x \textbf{ in } Q \textbf{ let } y = q'(x) \textbf{ where } F(x, y) \textbf{ return } Q''(x, y)$$

for the query

$$\bigcup_{z \in P} Q''(z_1, z_2) \qquad \text{with} \qquad P := \big\{z \in \{x * q'(x) \,:\, x \in Q\} \mid F(z_1, z_2)\big\}$$

of type $set(a'')$.

Example 6 (DL-style Queries). For a relation $\vdash_\Sigma R < a, a'$, a concept $c < a$, and a query $\vdash_\Sigma Q : set(a')$, we write $\Box^c R.Q$ for the query $\{x \in c \,|\, \forall y \in R(x).y \in Q\}$ of type $set(a)$.

Note that, contrary to the universal restriction $\Box R.Q$ in description logic, we have to restrict the query to all x of concept c instead of querying for all x of type a. This makes sense in our setting because we assume that we can only iterate efficiently over concepts but not over (possibly infinite!) base types.

However, this is not a loss of generality: individual signatures may always couple a base type a with a concept is_a such that $[\![is_a]\!] = [\![a]\!]$.

Declaration	Intuition
Base types	
$id\ : type$	URIs of MMT declarations
$obj\ : type$	MMT (OPENMATH) OBJECTS
$xml\ : type$	XML elements
Concepts	
$theory < id$	theories
$view < id$	views
$constant < id$	constants
$style < id$	styles
Relations	
$includes < id, id$	inclusion between theories
$declares < id, id$	declarations in a theory
$domain < id, id$	domain of structure/view
$codomain < id, id$	codomain of structure/view
Functions	
$type\ : id \to obj$	type of a constant
$def\ : id \to obj$	definiens of a constant
$infer\ : id, obj \to obj$	type inference relative to a theory
$arg_p\ : obj \to obj$	argument at position p
$subobj\ : obj, id \to set(obj)$	subobjects with a certain head
$unify\ : obj \to set(id \times obj \times obj)$	all objects that unify with a given one
$render\ : id, id \to xml$	rendering of a declaration using a certain style
$render\ : obj, id \to xml$	rendering of an object using a certain style
$u\ : id$	literals for MMT URIs u
$o\ : obj$	literals for MMT objects o
Predicates	
$occurs\ : id, obj \to prop$	occurs in

Fig. 8. The QMT Signature for MMT

3 Querying MMT Libraries

We will now fix an MMT-specific signature Σ that customizes QMT with the MMT ontology as well as with several functions and predicates based on the MMT specification. The declarations of Σ are listed in Fig. 8.

For simplicity, we avoid presenting any details of MMT and refer to [RK11] for a comprehensive description. For our purposes, it is sufficient to know that MMT organizes mathematical knowledge in a simple ontology that can be seen as the fragment of OMDOC pertaining to formal theories. We will explain the necessary details below when explaining the respective Σ-symbols.

An MMT **library** is any set of MMT declarations (not necessarily well-typed or closed under dependency). We will assume a fixed library L in the following. Based on L, we will define a model M by giving the interpretation s^M for every symbol s listed in Fig. 8.

Base Types We use three base types. Firstly, every Mmt declaration has a globally unique **canonical identifier**, its Mmt URI. We use this to define id^M as the set of all Mmt URIs declared in L.

obj^M the set of all OpenMath objects that can be formed from the symbols in id^M. In order to handle objects with free variables conveniently, we use the following convention: All objects in obj^M are technically closed; but we permit the use of a special binder symbol `free`, which can be used to formally bind the free variables. This has the advantage that the context of an object, which may carry, e.g., type attributions, is made explicit. Using general OpenMath objects means that the type *obj* is subject to exactly α-equality and attribution flattening, the only equalities defined in the OpenMath standard. The much more difficult problem of queries relative to a stronger equality relation remains future work.

The remaining base type *xml* is a generic container for any non-Mmt **XML data** such as HTML or presentation MathML. Thus, xml^M is the set of all XML elements. This is useful because the Mmt API contains several functions that return XML.

Ontology For simplicity, we restrict attention to the most important notions of the Mmt ontology; adding the remaining notions is straightforward. The ontology only covers the Mmt declarations, all of which have canonical identifiers. Thus, all concepts refine the type *id*, and all relations are between identifiers.

Among the Mmt **concepts**, **theories** are used to represent logics, theories of a logic, ontologies, type theories, etc. They contain **constants**, which represent function symbols, type operators, inference rules, theorems, etc. Constants may have OpenMath **objects** [BCC+04] as their type or definiens. Theories are related via theory morphisms called **views**. These are truth-preserving translations from one theory to another and represent translations and models. Theories and views together form a multi-graph of theories across which theorems can be shared. Finally, **styles** contain notations that govern the translation from content to presentation markup.

Mmt theories, views, and styles can be structured by a strong module system. The most important modular construct is the *includes* relation for explicit imports. The *declares* relation relates every theory to the constants it declares; this includes the constants that are not explicitly declared in L but induced by the module system. Finally, two further relations connect each view to its *domain* and *codomain*.

All concepts and relations are interpreted in the obvious way. For example, the set $theory^M$ contains the Mmt URIs of all theories in L.

Function and Predicate Symbols Regarding the function and predicate symbols, we are very flexible because a wide range of operations can be defined for Mmt libraries. In particular, every function implemented in the Mmt API can be easily exposed as a Σ-symbol. Therefore, we only show a selection of symbols that showcase the potential.

In Sect. 2, we have deliberately omitted **partial function symbols** in order to simplify the presentation of our language. However, in practice, it is often necessary to add them. For example, def^M must be a partial function because (i) the argument might not be the MMT URI of a constant declaration in L, or (ii) even if it is, that constant may be declared without a definiens. The best solution for an elegant treatment of partial functions is to use option types $\mathit{opt}(t)$ akin to set types $\mathit{set}(t)$. However, for simplicity, we make $[\![-]\!]$ a partial function that is undefined whenever the interpretation of its argument runs into an undefined function application. This corresponds to the common concept of queries returning an error value.

The partial **functions** type^M and def^M take the identifier of a constant declaration and return its type or definiens, respectively. They are undefined for other identifiers.

The partial function $\mathit{infer}^M(u, o)$ takes an object o and returns its dynamically inferred type. It is undefined if o is ill-typed. Since MMT does not commit to a type system, the argument u must identify the type system (which is represented as an MMT theory itself). If O is a binding object of the form $\mathtt{OMBIND}(\mathtt{OMS}(\mathtt{free}), \Gamma, o')$, the type of o' is inferred in context Γ.

arg_p is a family of function symbols indexed by a natural number p. p indicates the position of a direct subobject (usually an argument), and $\mathit{arg}_p^M(o)$ is the subobject of o at position p. In particular, $\mathit{arg}_i^M(\mathtt{OMA}(f, a_1, \ldots, a_n)) = a_i$. Note that arbitrary subobjects can be retrieved by iterating arg_p. Similarly, $\mathit{subobj}^M(o, h)$ is the set of all subobjects of o whose head is the symbol with identifier h. In particular, the head of $\mathtt{OMA}(\mathtt{OMS}(h), a_1, \ldots, a_n)$ is h. In both cases, we keep track of the free variables, e.g., $\mathit{arg}_2^M(\mathtt{OMBIND}(b, \Gamma, o)) = \mathtt{OMBIND}(\mathtt{OMS}(\mathtt{free}), \Gamma, o)$ for $b \neq \mathtt{OMS}(\mathtt{free})$.

$\mathit{unify}^M(O)$ performs an object query: It returns the set of all tuples $u * o * s$ where u is the MMT URI of a declaration in L that contains an object o that unifies with O using the substitution s. Here we use a purely syntactic definition for unifiability of OPENMATH objects.

$\mathit{render}^M(o, u)$ and $\mathit{render}^M(d, u)$ return the presentation markup dynamically computed by the MMT rendering engine. This is useful because the query and the rendering engine are often implemented on the same remote server. Therefore, it is reasonable to compute the rendering of the query results, if desired, as part of the query evaluation. Moreover, larger signatures might provide additional functions to further operate on the presentation markup. *render* is overloaded because we can present both MMT declarations and MMT objects. In both cases, u is the MMT URI of the style providing the notations for the rendering.

The **predicate** symbol *occurs* takes an object O and an identifier u, and returns true if u occurs in O.

Finally, we permit **literals**, i.e., arbitrary URIs and arbitrary OPENMATH objects may be used as nullary constants, which are interpreted as themselves (or as undefined if they are not in the universe). This is somewhat inelegant but necessary in practice to form interesting queries. A more sophisticated QMT signature could use one function symbol for every OpenMath object constructor instead of using OpenMath literals.

Example 7. An Mmt theory graph is the multigraph formed by using the theories as nodes and all theory morphisms between them as edges. The components of the theory graph can be retrieved with a few simple queries.

Firstly, the set of theories is retrieved simply using the query *theory*. Secondly, the theory morphisms are obtained by two different queries:

$$\begin{array}{ll} \text{views} & \{v * x * y \,:\, v \in \mathit{view},\ x \in \mathit{domain}(v),\ y \in \mathit{domain}(v)\} \\ \text{inclusions} & \bigcup_{y \in \mathit{theory}} \{x * y \,:\, x \in \mathit{includes}^*(y)\} \end{array}$$

The first one returns all view identifiers with their domain and codomain. Here we use an extension of the replacement operator $\{\_ : \_\}$ from Ex. 3 to multiple variables. It is straightforward to define in terms of the unary one. The second query returns all pairs of theories between which there is an inclusion morphism.

Example 8. Consider a constant identifier $\exists I$ for the introduction rule of the existential quantifier from the natural deduction calculus. It produces a constructive existence proof of $\exists x.P(x)$; it takes two arguments: a witness w, and a proof of $P(w)$. Moreover, consider a theorem with identifier u. Recall that using the Curry-Howard representation of proofs-as-objects, a theorem u is a constant, whose type is the asserted formula and whose definiens is the proof.

Then the following query retrieves all existential witnesses that come up in the proof of u:

$$\{\mathit{arg}_1(x) \,:\, x \in \mathit{subobj}(\mathit{def}(u), \exists I)\}$$

Here we have used the replacement operator introduced in Ex. 3.

Example 9 (Continuing Ex. 8). Note that when using $\exists I$, the proved formula P is present only implicitly as the type of the second argument of $\exists I$. If the type system is given by, for example, LF and type inference for LF is available, we can extend the query from Ex. 8 as follows:

$$\{\mathit{arg}_1(x) * \mathit{infer}(LF, \mathit{arg}_2(x)) \,:\, x \in \mathit{subobj}(\mathit{def}(u), \exists I)\}$$

This will retrieve all pairs (w, P) of witnesses and proved formulas that come up in the proof of u.

4 Implementation

We have implemented QMT as a part of the Mmt API. The implementation includes a concrete XML syntax for queries and an integration with the Mmt web server, via which the query engine is exposed to users. The server can run as a background service on a local machine as well as a dedicated remote server. Sources, binaries, and documentation are available at the project web site [Rab08].

The Mmt API already implements the Mmt ontology so that appropriate indices for the semantics of all concept and relation symbols are available. Indices scale well because they are written to the hard drive and cached to memory on demand. With two exceptions, the semantics of all function and predicate symbols is implemented by standard Mmt API functions.

The semantics of *unify* is computed differently: A substitution tree index of the queries library is maintained separately by an installation of MathWebSearch [KŞ06]. Thus, QMT automatically inherits some heuristics of MathWebSearch, such as unification up to symmetry of certain relation symbols. MathWebSearch and query engine run on the same machine and communicate via HTTP.

Another subtlety is the semantics of *infer*. The Mmt API provides a plugin interface, through which individual type systems can be registered; the first argument to $infer^M$ is used to choose an applicable plugin. In particular, we provide a plugin for the logical framework LF [HHP93], which handles type inference for any type system that is formalized in LF; this covers all type systems defined in the LATIN library [CHK+11] and thus also applies to our imports of the Mizar [TB85] and TPTP libraries [SS98].

Query servers for individual libraries can be set up easily. In fact, because the Mmt API abstracts from different backends, queries automatically return results from all libraries that are registered with a particular instance of the Mmt API. This permits queries across libraries, which is particularly interesting if libraries share symbols. Shared symbols arise, for example, if both libraries use the standard OpenMath CDs where possible or if overlap between the libraries' underlying meta-languages is explicated in an integrating framework like the LATIN atlas [CHK+11].

Example 10. The LATIN library [CHK+11] consists of over 1000 highly modularized LF signatures and views between them, formalizing a variety of logics, type theories, set theories, and related formal systems. Validating the library and producing the index for the Mmt ontology takes a few minutes with typical desktop hardware; reading the index into memory takes a few seconds. Typical queries as given in this paper are evaluated within seconds.

As an extreme example, consider the query $Q = Declares(theory)$. It returns in less than a second the about 2000 identifiers that are declared in any theory. The query $\bigcup_{x \in Q}\{x * type(x)\}$ returns the same number of results but pairs every declaration with its type. This requires the query engine to read the types of all declarations (as opposed to only their identifiers). If none of these are cached in memory yet, the evaluation takes about 4 minutes.

5 Conclusion and Future Work

We have introduced a simple, expressive query language for mathematical theories (QMT) that combines features of compositional, property, and object query languages. QMT is implemented on top of the Mmt API; that provides any library that is serialized as Mmt content markup with a scalable, versatile querying engine out of the box. As both Mmt and its implementation are designed to admit natural representations of any declarative language, QMT can be readily applied to many libraries including, e.g., those written in Twelf, Mizar, or TPTP.

Our presentation focused on querying *formal* mathematical libraries. This matches our primary motivation but is neither a theoretical nor a practical restriction. For example, it is straightforward to add a base type for presentation MathML and some functions for it. MathWebSearch can be easily generalized to permit unification queries on presentation markup. This also permits queries that mix content and presentation markup, or content queries that find presentation results. Moreover, for presentation markup that is generated from content markup, it is easy to add a function that returns the corresponding content item so that queries can jump back and forth between them.

Similarly, we can give a QMT signature with base types for authors and documents (papers, book chapters, etc.) as well as relations like `author-of` and `cites`. It is easy to generate the necessary indices from existing databases and to reuse our implementation for them. Moreover, with a relation `mentions` between papers and the type *id* of mathematical concepts, we can combine content and narrative aspects in queries. An index for the `mentions` relation is of course harder to obtain, which underscores the desirability of mathematical documents that are annotated with content URIs.

References

ADL12. D. Aspinall, E. Denney, and C. Lüth. Querying Proofs. In *Proceedings of LPAR*, 2012. To appear.

ANS03. ANSI/ISO/IEC. 9075:2003, Database Language SQL, 2003.

AS04. Andrea Asperti and Matteo Selmi. Efficient Retrieval of Mathematical Statements. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management*, pages 17–31. Springer, 2004.

AY08. M. Altamimi and A. Youssef. A Math Query Language with an Expanded Set of Wildcards. *Mathematics in Computer Science*, 2:305–331, 2008.

BCC$^+$04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

BR03. G. Bancerek and P. Rudnicki. Information Retrieval in MML. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Mathematical Knowledge Management*, pages 119–132. Springer, 2003.

CHK$^+$11. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.

GC03. F. Guidi and C. Sacerdoti Coen. Querying Distributed Digital Libraries of Mathematics. In T. Hardin and R. Rioboo, editors, *Proceedings of Calculemus*, pages 17–30, 2003.

HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

KRZ10. M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.

KŞ06. M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.

LM06. P. Libbrecht and E. Melis. Methods to Access and Retrieve Mathematical Content in ActiveMath. In A. Iglesias and N. Takayama, editors, *International Congress on Mathematical Software*, pages 331–342. Springer, 2006.

MG08. J. Mišutka and L. Galamboš. Extending full text search engine for mathematical content. In P. Sojka, editor, *Towards a Digital Mathematics Lbrary*, pages 55–67, 2008.

MM06. R. Munavalli and R. Miner. MathFind: a math-aware search engine. In E. Efthimiadis, S. Dumais, D. Hawking, and K. Järvelin, editors, *International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 735. ACM, 2006.

MY03. B. Miller and A. Youssef. Technical Aspects of the Digital Library of Mathematical Functions. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):121–136, 2003.

Rab08. F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.

RK11. F. Rabe and M. Kohlhase. A Scalable Module System. see http://arxiv.org/abs/1105.0548, 2011.

SL11. P. Sojka and M. Líska. Indexing and Searching Mathematics in Digital Libraries - Architecture, Design and Scalability Issues. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 228–243. Springer, 2011.

SS98. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

Urb06. J. Urban. MOMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

W3C03. W3C. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See http://www.w3.org/TR/MathML2.

W3C07. W3C. XQuery 1.0: An XML Query Language, 2007. http://www.w3.org/TR/xquery/.

W3C08. W3C. SPARQL Query Language for RDF, 2008. http://www.w3.org/TR/rdf-sparql-query/.

ZK09. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.