JACOBS
UNIVERSITY

Enabling Cross-System Communication Using Virtual Theories and QMT

by

Tom Wiesing

a thesis for conferral of a Master of Science in Data Engineering

Prof. Dr. Michael Kohlhase

Dr. Florian Rabe

Date Of Submission: August 15, 2017

Statutory Declaration

I, Tom Wiesing, hereby declare that I have written this thesis independently, unless where clearly stated otherwise. I have used only the sources, the data and the support that I have clearly mentioned. This thesis has not been submitted for conferral of degree elsewhere.

Bremen, August 15, 2017

Signature _____

Abstract

Mathematical Knowledge Systems (MKS) are software solutions used by mathematicians in practice to explore specific fields of mathematics and help solving both abstract and computational problems. To solve a specific problem multiple systems are often necessary.

Existing cross-system-communication solutions are ad-hoc, non-expandable, exist only for the most common system combinations and do not maintain semantics across systems. Mathematicians are not interested in technical details required to use these approaches – they only want to explore mathematics and solve problems. In this thesis, we aim to address this problem by designing, implementing and demonstrating a generic, efficient and scalable approach for enabling transparent, semantics-aware, distributed computation across MKS.

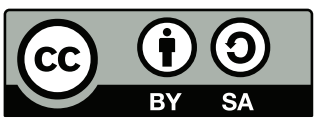
MMT is a language and framework that developed to manage mathematical knowledge. It makes use of the theory graph metaphor by organizing knowledge within different concrete theories that are represented by files on disk. To access knowledge in MMT, users can make use a Query Language called QMT.

In the OpenDreamKit project, an EU Horizon 2020 project aiming to provide tools for combining different existing mathematical software systems, we in particular want to make knowledge of these systems available to each other. For this we make use of the so-called Math-In-The-Middle (MiTM) paradigm, a new mathematics-aware, semantic, extensible approach to connecting multiple systems. Instead of building translations between any two of the involved systems, we decided to model the underlying mathematics in MMT and only build interfaces between this Math-In-The-Middle and the systems.

On top of the MiTM approach, the concept of virtual theories is introduced to MMT. These are just like concrete theories, but without the assumption of loading all declarations from a file on disk at system startup. We use them to transparently access knowledge across systems. Additionally, we expand the QMT Query Language to enable mathematicians to send queries in a system-independent manner.

In this thesis, we first recap the existing architecture, then move on to discuss our implementation of the Math-In-The-Middle Approach. Next we focus on the example of LMFDB, a mathematical database of objects in number theory and show how it can be implemented as a Virtual Theory to enable the desired communication and computation. Finally, we demonstrate the validity of our approach based on a concrete example.

Contents



1 Introduction

There are a multitude of different mathematical knowledge systems and tools. A mathematical knowledge system, or MKS, can be defined as a software solution which is aware of (a subset of) mathematical knowledge and can help mathematicians explore a specific field of mathematics.

These systems can range from calculators, which are only capable of performing simple computations, via mathematical databases (storing a set of a mathematical objects) to powerful modeling tools and computer algebra systems (CAS), that feature a broad variety of features. A few examples of common MKS include the databases like [Inc] and [LMF], systems like [GAP16] and [Dev], or the Wolfram Mathematica [Wol17] Computer Algebra System.

Most of these systems are very specific – they focus on one or very few aspects of mathematics. For example, OEIS is a database of integer sequences and nothing else, LMFDB is a database of objects in number theory. GAP excels at discrete algebra, whereas SageMath focuses on Algebra and Geometry in general.

For a mathematician however (a user; let us call her Jane) the systems themselves are not relevant, instead she only cares about being able to solve problems. Typically, it is not possible to solve a mathematical problem using only a single problem. Thus Jane needs to work with multiple systems and combine the results to reach a solution. Currently there is very little help with this practice, so Jane has to isolate sub-problems the respective systems are amenable to, formulate them into the respective input language, collect results, and reformulate them for the next system âĂŤ a tedious and error-prone process at best, a significant impediment to scientific progress in its overall effect. Solutions for some situations certainly exist, which can help get Jane unstuck, but these are ad-hoc and for specific, often-used system combinations only. Each of these requires a lot of maintenance and does not scale to a larger set of specialist systems.

In this thesis, we develop tools and methods of making mathematical software systems interoperable and contribute to a systematic, knowledge-based paradigm for system interoperability – the **Math-in-the-Middle (MitM) paradigm** developed in the OpenDreamKit project – a large-scale project that aims at providing a virtual research environment toolkit for mathematics based on existing (specialist) mathematical software systems.

In the rest of the introduction, we will briefly review the concepts of the MitM paradigm leading to our research question before we embark in the technical discussion.

1.1 The MMT System, Libraries and MathHub

The center of the MitM paradigm is the **Math-in-the-Middle ontology**, a flexiformal representation of mathematical knowledge, which is represented in the the OMDoc/MMT-language, mechanized by the MMT system, and stored in the MathHub information system.

MMT [Rab11] is a language and framework that was developed to manage (mainly) mathematical knowledge. It can handle both formal and informal content. MMT is under

active development and the project home page can be found at [MMTa].

Knowledge inside of MMT employs the theory graph paradigm supported by the underlying OMDoc/MMT language. This means that knowledge is organized in distinct theories which are related by different relations in a graph-like manner. One of these relations is a so-called view, which can be used to translate knowledge from one theory to another. Definiens within each theory are identified by a URI (the so-called MMT URI) whereas the definitions are represented by what amounts to OpenMath [Bus+04] objects. Apart from making purely formal declarations, it is also possible to annotate the declarations with human-readable, narrative elements.

Commonly all knowledge in MMT is manifested on disk and loaded by the system in its entirety on startup. This means that there is a set of files, specifically XML files, on disk containing the appropriate theories and declarations. We refer to theories stored in this form as *concrete theories*.

Knowledge in MMTcan be accessed by either giving it's name – i.e. it's MMT URI, or by giving a set of conditions that has to fulfilled by the knowledge in question. The achieve the latter, MMThas a Query Language called QMT, which allows even complex conditions to be specified.

This modeling of mathematical knowledge has proven very effective. On top of pure mathematical knowledge modeling, it is also possible to annotate declarations. This means one can give them human-readable descriptions. The other direction, giving informal knowledge (for example a paper) a partially formal meaning is also possible.

This has allowed us to develop and populate a system called MathHub [MH]. The system altogether forms a big theory graph totaling at about 10000 theories. MathHub is far more than just a theory graph however, it allows creating and editing of existing libraries and offers possibilities for integrating semantic services. The deatils of this are not relevant to the topic at hand – we refer the interested reader to [Ian17] instead.

Notable projects available on MathHub include SMGloM [SMG] – a semantic glossary of mathematical knowledge – LATIN [KMR] – a logical atlas of several different formalizations – as well as parts of the PVS [PVS], MiZar [Miz] and HOL Light [KR14] libraries.

1.2 Existing Ad-Hoc Solutions for System Interaction

With the exception of Mathematica none of the previously mentioned systems attempt to be universal – they do not try to model mathematics as a whole. Their focus on specific topics enables them to make concrete optimizations and provide detailed implementations of specific solutions.

As a result, these systems work very differently – for example databases only contain a fixed set of objects whereas computer algebra systems commonly allow for complicated manipulation of complex objects. This initially limits the inter actability of these systems, and makes it difficult for them to work together. Nonetheless, ad-hoc solutions exist and we proceed by giving the reader a brief overview.

The situation is best illustrated by an example. Consider again our user Jane. She wishes to check the hypothesis that all abelian transitive groups are cyclic. How can she

go about verifying this hypothesis experimentally with the help of different MKS?

Jane realizes that the LMFDB database contains a set of known abelian transitive groups. Furthermore, she realizes that the GAP system can check if a given group is cyclic. To check her hypothesis, she could first retrieve all relevant groups from LMFDB, and then use GAP to check if each of these is cyclic. While a positive result would not formally prove her claim, it would certainly give her experimental evidence and show that she should try to prove this formally.

How is such cross-system communication achieved in practice? LMFDB has implemented an export functionality of the database. Each time new data is available in LMFDB it is exported and included in the SageMath software distribution. Within SageMath, Jane is provided with a low-level API that can then be used to access this dump.

To communicate with GAP, a similar approach is chosen. A version of GAP is included within Sage, providing a GAP prompt. Again, Jane is provided with an interface to make use of GAP commands from within Sage.

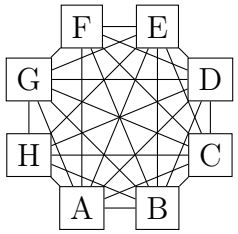


Figure 1: Common Ad-Hoc Approach to Connecting Systems.

Jane has to first use the LMFDB interface to find all abelian transitive groups. This means figuring out how LMFDB works and then manually transferring the objects from LMFDB to Sage. Next, she has to make sure that these objects are in a representation that GAP can understand. After some translation work, she can then open the GAP prompt in Sage and check for each group if GAP thinks it is cyclic.

This is obviously a very tedious approach that can easily lead to mistakes. It also suffers from many other problems.

Semantic-less interfaces When stating that LMFDB it is exported and included in the SageMath software distribution describes the level of integration well. The semantics of LMFDB are not maintained during this process.

For example, boolean values are commonly represented as integers 1 or 0 in LMFDB. When accessing the same value from SageMath, no conversion into a SageMath boolean (which is in fact just a python object) is performed – Jane still only sees a 1 or a 0. The difference between 1 and `true` may seem trivial, but this becomes an issue when talking about data types like matrices or vectors. Continuing, Jane needs to know that the `ab` of a group object in LMFDB is 1 if it is abelian and 0 if it is not.

This is not very mathematical, and it requires being familiar with the technical details of how groups are represented inside of LMFDB.

Updating Problem This obviously requires packaging both LMFDB and GAP for usage within Sage. Each time a change is made in either of these systems, they need to be re-packaged and Sage needs to be updated. This gives rise to a version management problem – the usage of the newest versions of LMFDB or GAP is not automatic – and a data duplication issue.

Scaling Up How is more complex communication managed?

> Imagine a user in GAP suddenly wanting to make use of LMFDB. This would mean having to add LMFDB to GAP as well. This not only means more data duplication, it also means having to build another bridge between two systems. In general, to enable communication between n systems each pair of systems need to be connected. This means $O(n^2)$ connections need to be built, leading to a situation as in Figure 1.

Mathematica, unlike the other systems mentioned here, attempts to be a universal system. One one hand this does have the advantage that it can handle a large subset of Mathematics at once. On the other hand, by nature of being a commercial product and not an OpenSource effort, it is not entirely clear how interactions between different components are achieved. Mathematica is usually not able to connect to other software systems and mathematicians using Mathematica are required to use only tools available natively in the system.

1.3 The Math-In-The-Middle Approach

Recall that the OpenDreamKit [ODKb] project is an EU-funded project that aims to create Virtual Research Environments enabling mathematicians to make efficient use of existing Open-Source mathematical knowledge systems. These systems include the aforementioned Sage, GAP and LMFDB systems, along with others.

We give a short overview of the OpenDreamKit project and involved systems here. For a a comprehensive list of all projects included in the OpenDreamKit project and in particular the Math-In-The-Middle approach we refer the interested reader to the OpenDreamKit project proposal [ODKa] as well as reports on the Math-In-The-Middle approach [Deh+16].

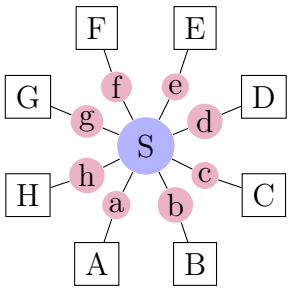


Figure 2: The MiTM Approach to Connecting Systems.

For this thesis, we build on and make use of the Math-In-The-Middle approach, which can be seen in Figure 2. It models the true, underlying mathematical semantics in MMT (blue S) and allows translation between this centrally formalized knowledge and the systems on the boundary (capital letters A - H). This mathematical knowledge is modeled using the well-established theory graph paradigm and is stored inside our MathHub / MMT architecture.

The knowledge in the systems themselves is also modeled via theories in a theory graph (small letters a - h). This allows us to implement translation with the help of bi-views – views in both directions – between the central knowledge – the Math-In-The-Middle – and each of the systems.

Using this mechanism leaves the semantics of translated objects intact – thus solving the first of the disadvantages listed above. Furthermore, when adding a new system it is only required to add 2 new translations, one from the central math ontology to the system and one in the other direc-

tion. This drastically reduces the scaling problem – suddenly only $O(n)$ translations are needed when connecting n different systems.

1.4 Research Question

This leads to our research question. How can different mathematical knowledge systems be connected in a generic, efficient and scalable manner to enable transparent, semantics-aware, distributed computation?

As we have seen, to solve this question and be semantics-aware it is required to model mathematics independent of the different systems. We need to separate the representations of objects inside each of the systems from their mathematical meaning, the semantics. This is achieved by using the semantics-aware, scaleable Math-In-The-Middle approach.

However, the Math-In-The-Middle approach only provides part of the answer to this question. A complete solution needs to address three more aspects which are:

(i) a generic mechanism to manage knowledge stored in the systems and keep it in sync with the Math-In-The-Middle ontologies,

(ii) a system-independent query language, capable of allowing users like Jane to transparently formulate computational problems to be solved, and

(iii) an efficient lower-level communication layer capable of transporting computational tasks and results between systems, allowing the computational tasks defined by the query language to be resolved in a distributed manner.

In the following, we give a short overview of how our contribution addresses each of them.

1.5 Contribution: A Knowledge-based Framework for Distributed Computation at the Math-in-the-Middle Level

Recall that QMT is the Query Language of MMT. As such, it was designed to formulate queries about any kind of knowledge inside of theory graphs and especially inside of OMDoc/MMT. It is thus well-suited for aspect (ii) above – it can be used to formulate queries in a system-independent fashion. During this thesis, we have extended it to better represent computational aspects and improved support for generic knowlege management.

SCSCP [Fre+09] stands for Symbolic Computation Software Composability Protocol. It is an Open Standard which has been adopted by the OpenMath Society and is already in use by some of the systems in the OpenDreamKit project. To enable communication between different mathematical systems it uses a Remote Procedure Call Model. All objects and procedure calls are represented using OpenMath objects, making it ideal for out situation – all content is already modeled in OMDoc/MMT using OpenMath. Thus we can make use of it to implement aspect (iii).

External Knowledge and the Concept of Virtual Theories

This leaves just aspect (i) unsolved. Knowledge inside of MMT comes from different sources. Either it is specifically created by authors or, like here, originally resides in some external system like another MKS.

Commonly, we use an Import/Export metaphor to make such knowledge available to MMT. This means we create an exporter from the external system that turns the knowledge representation of the external system directly into OMDoc/MMT. This knowledge is then stored by the MMT system in a set of XML files representing OMDoc/MMT. These XML files are then used by MMT just like any other concrete theory is used.

Such a process has proven to be very useful – it makes all information available to MMT directly. In other words, the knowledge is consolidated on the users disk – and thus it can be accessed even if the original system no longer works.

This means that any change in the external system requires us to re-import the knowledge – the same updating problem encountered in Jane's original approach above. This is not a scalable solution and thus does not suffice to address this aspect of our research question.

Instead, we have introduced a second type of theory to MMT, the so-called virtual theory. In a nutshell, virtual theories are just like concrete theories, but without the assumption of loading all declarations from a file on disk at system startup. Instead of loading all knowledge from an XML file, virtual theories load declarations in a lazy fashion when they are required. Here we do not even restrict ourselves to lazily reading an XML file, on the contrary, in most use cases we actually create the OMDoc/MMT representation on demand.

Notice that a replacement of the Import/Export metaphor by virtual theories only makes sense in certain situations. In the case of LMFDB, we can easily translate a single declaration, whereas in the case of a theorem prover library like PVS this is not the case. Instead of having to start an entire theorem prover system to only load a single declaration, it makes far more sense to import the library as a whole.

In the architecture needed to solve the research question, it does make sense to use virtual theories. virtual theories have the key advantage that we can directly access the knowledge of other systems. There is little to no need to manually keep the Math-In-The-Middle ontology in sync – by use of a virtual theory this happens automatically.

Our core example for virtual theories is the the LMFDB database introduced above. An LMFDB virtual theory can retrieve the content of a single declaration in LMFDB form and proceed to translate it to OMDoc/MMT. Only items explicitly needed by the user or some other process inside of the MMT API need to be retrieved and translated.

The choice of virtual theories as a solution to aspect (i) also has implications for the query layer. As it turns out the existing QMT implementation relies on implementation details of concrete theories. In particular, it makes the assumption that all queryable knowledge is available inside of MMT memory. As this does not hold for virtual theories, during the course of this thesis we have had to adapt the implementation accordingly.

Solving Jane's Example

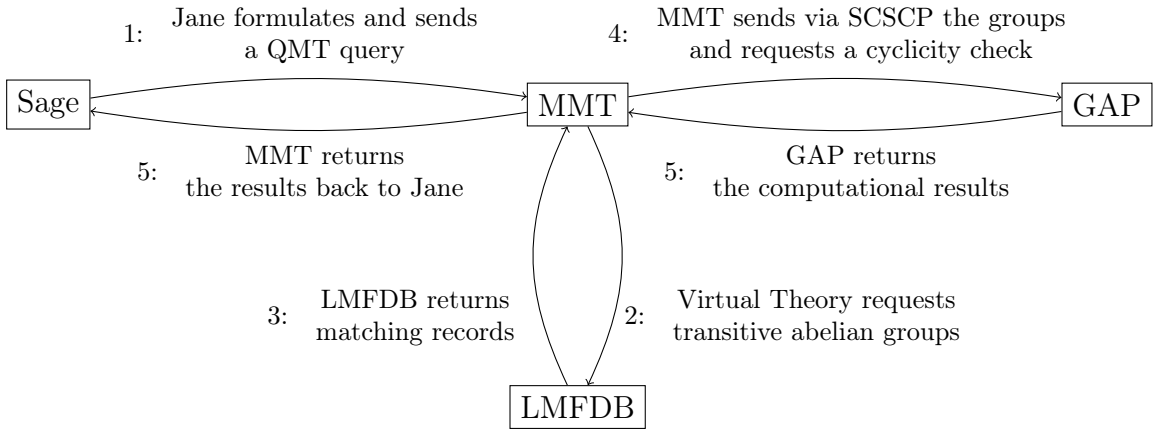


Figure 3: Solving Jane's example of Cross-System Communication using our approach.

Recall the example involving Jane from above. We give a short preview in Figure 3 of how our choices to use virtual theories, QMT and SCSCP can be used to solve this use-case.

Instead of writing all queries in a Sage-specific manner within the Sage System and targeting LMFDB and GAP directly, Jane can make use of the QMT Query Language and send a single query to MMT that expresses the semantics of the information she wishes to retrieve.

Unlike in existing solutions, she does not have to be aware of the low-level implementation details. Instead she can semantically ask for "abelian groups" and if a system is "cyclic" or not. The MMT system will figure out the exact system-specific details using views. Furthermore, all the intermediate steps are hidden from Jane's point of view – she can get her result in a single step instead of the multitude of steps required usually.

We will come back to this example later – and discuss it in more detail.

1.6 Structure

We will proceed as follows: In Section 2 we describe the structure of the MMT system, and then continue in Section 3 to describe the Math-In-The-Middle approach and how to utilize it with the Systems found in the OpenDreamKit project. We move on in Section 4 to detail our virtual theory implementation and how they function as a uniform conceptual interface to Mathematical Databases. Next, in Section 5 we examine the existing implementation and improvements we have made to the QMT Query Language and then discuss in Section 6 how it is used to enable cross-system communication within our architecture. Finally we conclude in Section 7 by revisiting Jane's example and discussing how the progress made during this thesis can be used in cross-system communication. We also give a brief outlook.

2 Overview of the MMT System

MMT [Rab11] is a language and framework that was developed to manage (mainly) mathematical knowledge and can handle both formal and informal content. It is foundation independent, i.e. it does not enforce a certain logic for formal content.

It consists of two main parts, the OMDoc/MMT language and the MMT System. The OMDoc/MMT language [Rab09] forms the basis of knowledge representation. The MMT System [MMTb], sometimes also referred to as the MMT API, is the implementation of this language written in Scala. It can be extended by custom plugins. The project is under active development and the source code can be retrieved via [MMTa].

2.1 Theories as Sets of Declarations

Knowledge in OMDoc/MMT is organized in theories which group knowledge together semantically. They consist of a set of declarations, each of which has a name as well as optionally a type, a definition and other kinds of meta data. A simple theory about Semigroups can be found in Figure 4.

Semigroup : LF		
G	:	`type`
$\circ$	:	$G \to G \to G$
assoc	:	`ded` $(\forall x \in G.\forall y \in G.\forall z \in G.(x \circ y) \circ z = x \circ (y \circ z))$

Figure 4: A Theory of Semigroups.

We start by giving the theory a name – Semigroup in this case – and declaring the logical foundation we are using to formalize it – in this case the LF Logical Framework [Pfe91]. The logical foundation used is called a meta-theory and is in itself an OMDoc/MMT theory[1].

In MMT, each theory resides within a specific namespace – which is not explicitly shown here. The namespace – for example `http://www.opendreamkit.org/algebra` – together with the theory name `Semigroup` forms the MMT URI of the theory `http://www.opendreamkit.org/algebra?Semigroup`. This URI can be used to refer to the theory.

To prevent having to write down long urls when they are used frequently MMT supports the concept of URI abbreviations. These function in a straightforward fashion. For example, instead of constantly having to write `http://www.opendreamkit.org/` one define an abbreviation `odk:` and then reduce URIs like `http://www.opendreamkit.org/algebra?Semigroup` to `odk:algebra?Semigroup`.

[1] In contrast to normal theories, meta-theories like LF within the MMT system are commonly implemented in the form of code, rather than just taking the form of declarations. This allows implementation of type checking as well as other MMT features.

A list of common namespace abbreviations can be found in Table 5. From this point forward, any URI makes use of these abbreviations.

Abbreviation	URI
meta:	http://cds.omdoc.org/meta
qmt:	http://cds.omdoc.org/qmt
odk:	http://www.opendreamkit.org/
mitm:	http://mathhub.info/MitM/
lmfdb:	http://www.lmfdb.org/

Table 5: A list of URI abbreviations used in MMT.

We continue by making three declarations. These represent the requirements for semigroups: A set of elements, a (closed) operation on them and the axiom that the operation is associative. G, which is declared as a type, represents the set of elements. Declaring it as a type allows us to use it in the declarations below.

Note that, like above, the name together with the URI for the theory provides a URI to the content being defined. Here this is `odk:algebra?Semigroup?G`. MMT URIs to declarations in general consist out of a triple (d, m, s). d is a path to a namespace[2] (here `odk:algebra`), m is the name of a theory[3] (here `Semigroup`) , and s is the name of a symbol (here G)

Most of the time MMT URIs are straightforward – however in certain situations are not. Principally MMT allows arbitrary nesting of theories – that is theories within theories.

Next, $\circ$ is declared and assigned the type $G \to G \to G$. Informally, this declares $\circ$ as a function taking two elements of G and returning another element of G. This makes it a closed operation on the semigroup elements. Finally, we declare the axiom of associativity called `assoc`. Here we make use of the *ded* symbol via the Curry-Howard isomorphism[How80]. Intuitively, the symbol returns the type of all proofs for the statement in question. Since we are declaring a constant of this type, this means that there is a proof for associativity – making it an axiom.

Types and definitions of objects make up the object layer. MMT objects, also called terms, are essentially OpenMath [Bus+04] objects with some custom extensions. As such, they can consist of

- references to symbols (for example, we just use $\circ$ above, but a formal system needs to know what $\circ$ is),
- literals (mostly Integers like $1, 2, \ldots$, but also strings),
- variables and constants,
- applications of terms to other terms ($G \to G$ is really the symbol $\to$ applied to two arguments G and G), and
- bindings of variables inside of terms (for example in terms like $\forall X.P(X)$, the variable X is bound inside of the $\forall$ term).

[2] d as in document. These are essentially namespaces – but like theories form groups of semantically related declarations so do documents form semantic groupings for theories.

[3] The m here stands for module – a theory or a view. We will define what a view is later on.

MMT terms can be parsed both from an XML syntax as well as from a human-readable and writable surface syntax. The surface syntax can be further defined by so-called notations. These are additional meta-data on declarations that specify how a specific symbol or application of a symbol should look like.

2.2 Structure of Theory Graphs

Theories can be related via three different relations, imports, structures, and views. Imports make knowledge, i.e. all declarations, from one theory available in another and are used to extend theories. Views are truth-preserving mappings from declarations in one theory to another. Structures are very similar to views, except that they allow for renaming of declarations. Together Theories, Imports and Views naturally form a graph-like structure call a Theory Graph. An example of a theory graph – extending the example above – can be found in Figure 6.

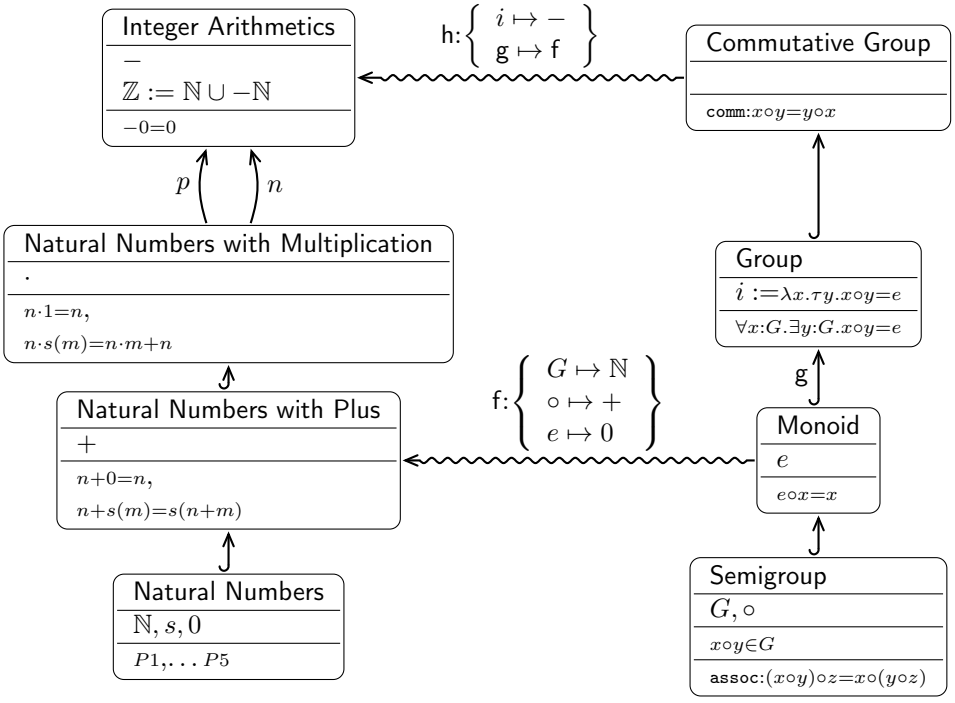


Figure 6: Simplified example of a theory graph. Imports and Structures are represented as solid edges and views as wavy edges. The mappings of the views are given explicitly. We omit the complete definitions of the Peano axioms for the natural numbers.

We start on the bottom right by declaring a semigroup as above. Due to complexity of the theory graph, we use more simplified notation and omit some details like meta-theories. In Mathematics, a Monoid can be defined as a semigroup together with an identity element. This is exactly what happens here as well, using an Import to from the

semigroup theory into the Monoid theory and then adding declarations for the identity element e. We continue with this principle, first we extend the Monoid into a Group by adding an identity element i together with an appropriate axiom and finally we extend this into a commutative group (top right) by adding an axiom of commutativity.

We also make use of the other relation between theories, a so-called view. This is done on the left hand side of the figure. Here we start on the bottom by defining the natural numbers, as formalized by the Peano Axiomes[4]. The next important step is the addition of a plus operation, again through the principle of importing the existing theory and adding another declaration.

Any skilled mathematician will have noticed that Natural Numbers together with addition are an example of a Monoid, in other words one can "view" the natural numbers as a Monoid. To express this formally, one has to be a bit more explicit. The set of elements G correspond to the natural numbers $\mathbb{N}$, the operation $\circ$ corresponds to $+$ and the identity element e is given by 0. This mapping is given by the view g and is truth-preserving in a certain sense: Everything that is true based on the axioms inside the $Monoid$ theory still holds in our Natural Numbers example – after translation via the view of course.

On the top left hand corner we make use of this mechanism even more. We would first like to make use of the Imports mechanism to create a theory of "Integer Arithmetics". But there is a problem: We need the natural numbers twice, one for positive numbers, once for negative numbers. To achieve this, we make use of structures, called p and n here. p imports the natural numbers as positive integers, and n imports them again as negative numbers. We then need to add the axiom that the zeros from both imports are actually the same.

Furthermore, we use the view h to show that this is an example of a "Commutative Group". The first line of the mapping is not too interesting – we just state the inverse is given by the negation of an integer – but something special happens in the second line. Note how we have not yet given mappings for a lot of the elements, for example groups still require a base set G which is formally known because of the Import. Instead of giving this mapping again, we instead take the import of the group theory g and map it via an existing view f. Similar to how theories are modularized via Imports, this allows us to structure views in a similar fashion.

2.3 Accessing Content in MMT by URI

The primary method for accessing knowledge in MMT is by name. Recall that all content, including theories, views and declarations have been assigned a URI. These URIs function like a name – each of them identifies a piece of content inside of MMT.

Consider for example, an author wishing to retrieve the associativity axiom for the *Semigroup* theory we defined in Example 6. Assuming that the author is familiar with our naming scheme, they can identify the URI `odk:algebra?Semigroup?assoc`. They can then directly access this URI in MMT.

[4] For simplicity, we omit their exact description here. They are merely referred to as p_1, p_2, p_3, p_4, p_5.

Another example of accessing a knowledge item comes from the OpenDreamKit project. During the course of the project, we have in particular made several curves stored within the LMFDB database available to MMT. Consider the elliptic curves dataset. Within MMT, this has been given the URI `lmfdb:db/elliptic_curve`. Each curve has a label assigned to by LMFDB itself, for example `11a1`. Should a user want to retrieve information about this specific curve, they can us the URI `lmfdb:db/elliptic_curve?11a1`.

2.4 Using URIs for Knowledge Representation

Consider the `Monoid` theory from Figure 6. Imagine we want to retrieve the associativity axiom for Monoids – can we just ask for the URI `odk:algebra?Monoid?assoc`? The immediate answer is no – there is no such declaration as it is only available via the *Import*. Thus the retrieval should fail.

In this example, one can easily retrieve the `odk:algebra?Semigroup?assoc` instead. We are familiar with the theory graph structure and can just follow the import. In other situations, the theory graph structure might not be as well-known by the person wanting to retrieve a declaration – or it might not be obvious. This is the case for the *Integer Arithmetics* theory – here we are "importing" the same theory twice via two different structures. Even if we want to look for the associativity axiom in this theory and made *Imported declarations* available it would not be clear how to make a difference between the two different associativity axioms.

MonoidFlat : LF		
G	:	`type`
$\circ$	:	$G \to G \to G$
`assoc`	:	`ded` $(\forall x \in G. \forall y \in G. \forall z \in G. (x \circ y) \circ z = x \circ (y \circ z))$
e	:	G
`ident`	:	`ded` $(\forall x \in G x \circ e = e \circ x = x)$

Figure 7: An elaborated, flattened version of the Monoid theory from Figure 6.

This is where the concept of flattening steps in. Flattening formalizes what we have only explained in a hand-wavy fashion so far – taking an imported (or structurally made available) theory and replacing it with explicit declarations. The process of flattening is sometimes also referred to as elaboration and an example can be found in Figure 7. We can see that this theory no longer has an import of the *Semigroup* theory – instead we have the declarations $G, \circ$ and *assoc*. Such theories are never written explicitly but instead are built by the MMT system internally – thus allowing us to retrieve `odk:algebra?Monoid?assoc` after all.

In summary, URIs are very useful – both in flattened and non-flattened form – when it comes to knowledge representation. This is only the case if one knows the structure of the knowledge one is talking about, as they enable us retrieve a specific knowledge item.

3 Utilizing the Math-In-The-Middle Approach

Recall that we want to make use of the Math-In-The-Middle approach to enable communication between different systems. The reasons for choosing the Math-In-The-Middle approach are first re-iterated, to then better explain the utilization of it.

There are two common approaches to make knowledge from one mathematical system B available in another system A:

- Accessing Bs API from A and translating the results, and
- Copying all knowledge available in B and integrating it into A.

If one uses the first approach, one has to rely on the B being constantly available. Furthermore, because A suddenly has to read knowledge from B, it has to be made aware of how the knowledge in B is represented. Thus if B is updated, or new features are added, the integration is likely to break.

The second approach is also subject to this problem, but to a much bigger extent. Not only does A have to be made aware of the knowledge structure of B, it is also required for developers to find a way to integrate the knowledge as a whole into the system. If B is updated, this approach commonly fails to scale, as it might become necessary to re-integrate the entire set of knowledge. This can lead to a situation where A entirely absorbs B, and future updates are only made within B. While this can be a good thing, it also gives less value to B and places a burden on the developers of A.

3.1 The Math-In-The-Middle Approach

In MMT, we commonly choose this second approach – that is we make use of an Import/-Export metaphor. We have already explain this in Section 1.5 and we briefly reiterate it here to allow the reader to better understand the Math-In-The-Middle approach.

We commonly create an exporter from the external system that turns the knowledge representation of the external system directly into OMDoc/MMT. This knowledge is then stored by the MMT system in a set of XML files representing OMDoc/MMT. These XML files are then used by MMT just like any other concrete theory is used.

Both approaches are one-way and not scaleable. If one wishes to reverse the integration, one has to build a completely new integration. Furthermore, to extend this model to n different systems that all make their knowledge available to each other, one has to build connections between any two. This results in $n(n-1) = O(n^2)$ total connections having to be built. Recall that this can be seen in Figure 1. This does not scale well with a large number of systems.

Furthermore recall that we model the true, underlying mathematics inside of MMT along with the knowledge contained in the systems themselves. Making use of theory graphs, this allows us to model translation between this knowledge and the different systems using views.

We call this approach the Math-In-The-Middle approach (or MiTM for short) and it directly solves the scalability problem mentioned above. When adding a new system, we no longer need to build translations between the new system and all other systems, instead we only need to model the knowledge with the help of the Math-In-The-Middle ontology. Thus for a set of n systems, we only need to implement $2n$ translations, one from and one to the underlying mathematical knowledge for each of the systems. Recall that this can be seen in Figure 2.

3.2 Systems in the OpenDreamKit Project

Recall that the OpenDreamKit [ODKb] is an EU-funded project that aims to create Virtual Research Environments enabling mathematicians to make efficient use of existing Open-Source mathematical software systems. We briefly introduce the main systems used in the OpenDreamKit project, as these are the use-case for cross-system communication.

For more details, as well as a comprehensive list of other projects in the project and in particular the Math-In-The-Middle approach we refer the interested reader to the Open-DreamKit project proposal [ODKa] as well as reports on the Math-In-The-Middle approach [Deh+16].

LMFDB The LMFDB [LMF] project is a mathematical database. It is implemented in Python with a MongoDB backend. The project contains several thousand L-Functions and curves as well as their properties. These mathematical objects are stored inside of several sub-databases, for example the elliptic curves database or the galois groups database [5].

GAP The GAP system [GAP16] is a computer algebra system. It is written in C. The abbreviation stands for "Groups, Algorithms and Programming" and the system focuses on content from group theory.

SageMath SageMath [Dev] is a project which has built a mathematical knowledge system written in Python. It combines multiple pre-existing projects and allows them to interact via a common interface using Python.

OEIS The On-Line Encyclopedia of Integer Sequences [Inc] is a website that collects and indexes sequences of integers. Each sequence is represented by a text file containing informally specified key-value pairs. The entire dataset encompasses about 280000 hand curated sequences.

3.3 The Math-In-The-Middle Ontology

We give a short overview of the structure of the Math-In-The-Middle ontology and how it is created. An overview can be seen in Figure 8.

[5] We will go into more details on this in Section 4.1 where LMFDB will serve as one of our main examples for virtual theories.

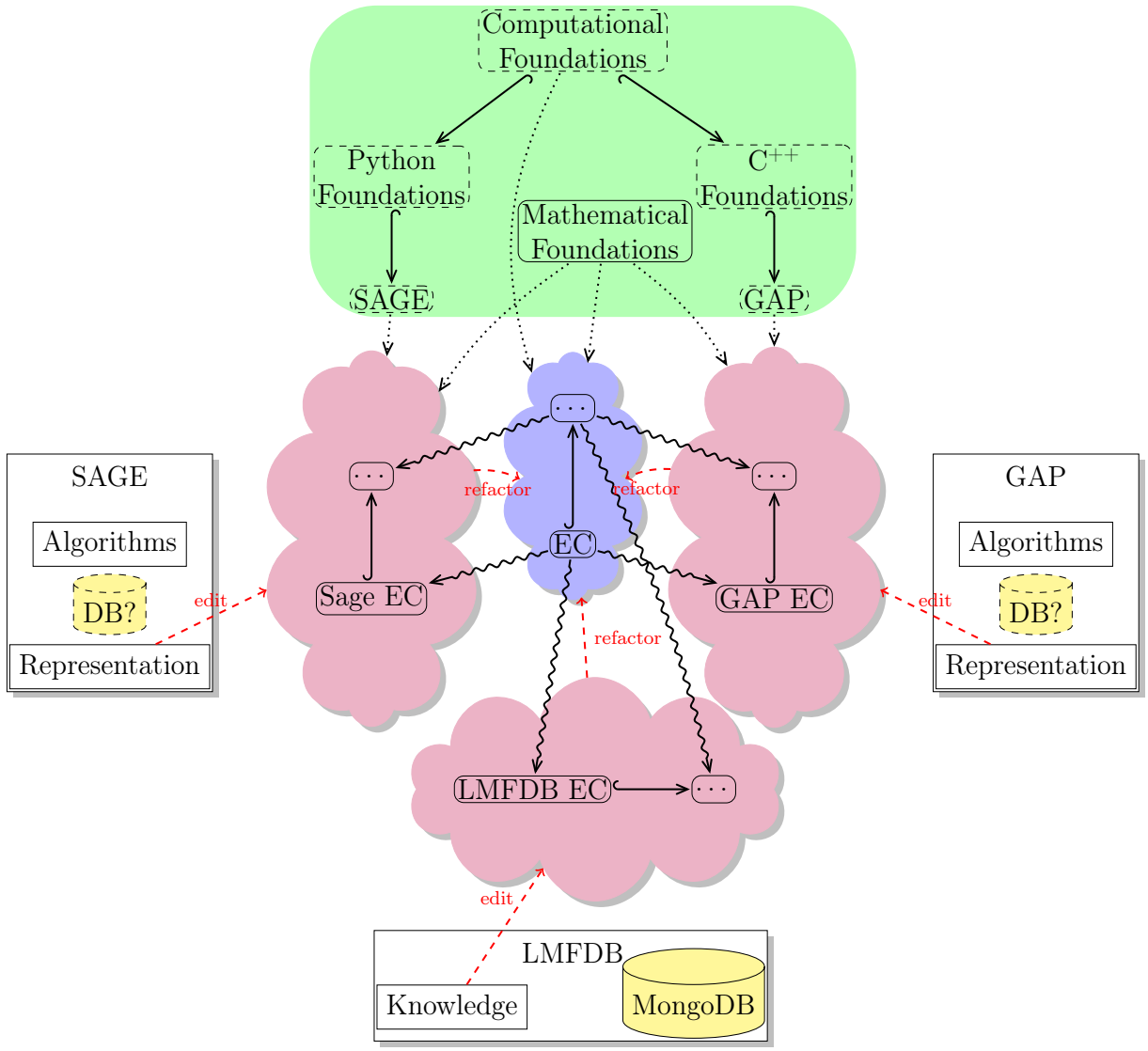


Figure 8: An overview of the MiTM ontology and how it interplays with the different systems.

It illustrates the Math-In-The-Middle ontology and its interaction with three of the involved systems – Sage, LMFDB and GAP. Furthermore, we only show the concept of elliptic curves (or EC for short). This has been chosen because it can be found in all three systems in a well-developed form.

Recall that everything is represented as a theory graph – each rounded box inside the image represents a theory. Includes are shown as solid edges, meta-theory relations as dotted edges and views as wavy edges. Everything else represents explanation elements and are not part of the theory graph itself.

The ontology is split into three parts.

Foundations As the title suggests, this part of the MiTM ontology forms the basis for the others. In the figure it is colored in green. This part of the ontology defines the basic data types and knowledge types needed to formalize the other parts. This in itself it split up in two parts, the mathematical foundations, and the system / programming language specific computational specific foundations.

On one hand, the mathematical foundations provide the basic mathematical objects needed – logics and booleans, as well as arithmetics and (real) numbers. This also includes a basic representation of Quantity Expressions such as $5\frac{\mathrm{m}}{\mathrm{s}}$.

On the other hand computational foundations define the programming language equivalent of these, such as boolean values and IEEE floats. On top of these, we define foundations for each of the systems. This is again a layered approach, we start by defining the foundations of the given programming languages, e.g. in the case of python defining how inheritance works. We then continue by formalizing the system foundation on top of this. For example, we define how filters[6] in GAP function.

Mathematics This part of the MiTM ontology is the largest and colored in blue. It contains the mathematical representation of objects that used by the systems. This is the core idea behind Math-In-The-Middle, and as such they are not based on any specific system foundation, but instead use only the system independent mathematical and computational foundations.

In the figure, it can be seen that the mathematical concept of elliptic curves is defined. Although not explicitly shown, this makes use of other concepts within the Math of MiTM.

System Interfaces This part of the ontology is the one most related to the systems themselves. For each of the systems involved, a separate set of theories exists. Each of these are shown in a separate purple cloud.

Consider for example the Sage System Interface (in purple on the left). This defines a Sage specific representation of elliptic curves along with a view from the mathematical representation to it There is actually a partial view back from `SageEC` back to the mathematical `EC`. This is not shown here, but together with the depicted view this

[6]In rough terms, filters are the GAP equivalent of classes.

enables bi-directional translation of objects in Sage and objects in the Math part of the ontology. This representation has the Sage Foundation as a meta theory, and can thus make use of Sage specific concepts.

A similar situation is the case for the GAP system, here of course we make use of the GAP meta-theory instead. This is different for LMFDB however. Notice that there is no existing foundation for it. This is because unlike the other systems, LMFDB consists only a database and thus never performs any kind of computation and thus no computational foundation is needed. Nonetheless, an interface theory for LMFDB is needed.

But how is this ontology created and maintained? This is key to solving aspect (i) of our research question. Consider the systems shown in the figure. With the exception of LMFDB each of the systems contains a representation of the knowledge along with a set of (computational) algorithms of that knowledge and some form of database of objects[7] .

Whenever the representation of this knowledge is changed, it is necessary to update the system interface ontology. This is represented by the red arrows in the figure. Furthermore, the central mathematical representation needs to be able to express objects of all three involved systems. Hence to create this representation, one refactors the three representations into a single central one. This is again expressed by red arrows.

3.4 Overview of the SCSCP Protocol

SCSCP [Fre+09] stands for Symbolic Computation Software Composability Protocol. It is a protocol that enables communication between different mathematical systems and is based on a remote procedure call model. Clients can send calls to servers and each server has a fixed set of methods that are exposed to clients.

In the context of our research question and the Math-In-The-Middle approach, the SCSCP protocol is used to facility communication between systems – addressing aspect (iii). SCSCP is an Open Standard that and has been adopted by the OpenMath Society.

All calls and values are represented as OpenMath objects. This makes it ideal for the MiTM approach – recall that our formalization makes use of theory graphs which in this case already represent objects as OpenMath objects. Client and servers communicate by exchanging a combination of XML processing instructions and XML-encoded OpenMath objects via sockets.

Each call to the SCSCP Server includes a reference to the function symbol being applied and a set of parameters for it. Furthermore, every call can include a set of options that specify how the server should treat the result. It can either be sent back to the client on completion, stored on the server for reference in future calls, or discarded entirely. An example of an SCSCP call can be seen in Figure 9 – it shows and XML representation of the OpenMath SCSCP call. Details are not shown here – instead the interested reader is referred to [D3.317] for complete technical details.

[7]LMFDB is a database with a MongoDB backend and thus does not contain a computational component.

As MMT implements theory graphs and thus OpenMath objects[8], it is ideal for implementing the SCSCP protocol.

During the progress of this thesis, an SCSCP Server and Client have been implemented in MMT, both written in Scala. The client functionality allows MMT to first encode an MMT term as an OpenMath object, then have any SCSCP server perform a computation on this object, and finally receive the response. This response can then be translated back into an MMT term. An example of the client functionality can be found in Figure 10 The server functionality allows MMT to expose computational functionality to other systems. Again, the server first receives an OpenMath object, translates it into an MMT term, performs some computation, and then returns the result as an OpenMath object.

[8] As mentioned, MMT objects are OpenMath objects plus some custom extensions. This means that some MMT objects are not strictly OpenMath objects and vice-versa. For this reason, a standards-compliant OpenMath object layer has been implemented. This enables efficient translation between MMT terms and OpenMath objects within Scala.

```
<OMOBJ>
  <OMATTR>

    <!-- First a set of options for the function call -->
    <OMATP>

      <!-- some per-session unique idenfier -->
      <OMS cd="scscp1" name="call_id" />
      <OMSTR>fa99c557370a87b83a4ac16e10c43940</OMSTR>

      <!-- Tell the server that it should return the result of the
         computation as an object -->
      <OMS cd="scscp1" name="option_return_object" />
      <OMSTR></OMSTR>
    </OMATP>

    <OMA>

      <!-- indicate that a procedure call is being made -->
      <OMS cd="scscp1" name="procedure_call" />

      <OMA>

        <!-- the procedure to call -->
        <OMS cd="scscp_transient_1" name="Addition"/>

        <!-- and the arguments to give to it -->
        <OMI>1</OMI>
        <OMI>1</OMI>
      </OMA>
    </OMA>
  </OMATTR>
</OMOBJ>
```

Figure 9: An (annotated) OpenMath representation of an SCSCP function call to compute and return the result of $1 + 1$.

```
// import the SCSCPClient and OpenMath libraries
import info.kwarc.mmt.odk.SCSCP.Client.SCSCPClient
import info.kwarc.mmt.odk.OpenMath._

// establish a connection
val client = SCSCPClient("scscp.gap-system.org")

// get a list of supported symbols
/** List(OMSymbol(Size,scscp_transient_1,None,None),
 * OMSymbol(Length,scscp_transient_1,None,None),
 * OMSymbol(LatticeSubgroups,scscp_transient_1,None,None),
 * OMSymbol(NrConjugacyClasses,scscp_transient_1,None,None),
 * OMSymbol(AutomorphismGroup,scscp_transient_1,None,None),
 * OMSymbol(Multiplication,scscp_transient_1,None,None),
 * OMSymbol(Addition,scscp_transient_1,None,None),
 * OMSymbol(IdGroup,scscp_transient_1,None,None),
 * ...,
 * OMSymbol(NextUnknownGnu,scscp_transient_1,None,None)) */
println(client.getAllowedHeads)

// We make a simple example: Apply the identity function to an integer 1
val identitySymbol = OMSymbol("Identity", "scscp_transient_1", None, None
    )
val identityExpression = OMApplication(identitySymbol, List(OMInteger(1,
    None)), None, None)

/** OMInteger(1,None) */
println(client(identityExpression).fetch().get)

// We also try to compute 1 + 1
val additionSymbol = OMSymbol("Addition", "scscp_transient_1", None, None
    )
val additionExpression = OMApplication(additionSymbol, List(OMInteger(1,
    None), OMInteger(1, None)), None, None)

/** OMInteger(2,None) */
println(client(additionExpression).fetch().get)

// and close the connection
client.quit()
```

Figure 10: This example shows use of the MMT SCSCP client to communicate with an external SCSCP server. In this case, the SCSCP server communicates with a server provided by the GAP system.

4 Virtual Theories as a Uniform Conceptual Interface for Mathematical Databases

As has been shown, it is necessary in the Math-In-The-Middle approach to create interface theories that model the knowledge in specific theories. Recall that one approach to achieving this is to export the knowledge and represent is as OMDoc/MMT, the language powering the MMT system. Such an approach generates a set of XML files that are loaded by the MMT system on startup.

Recall that we instead introduced a second type of theory to MMT, the so-called virtual theory. virtual theories are just like concrete theories, but without the assumption of loading all declarations from a file on disk at system startup.

4.1 The Structure of LMFDB

As a running example of a virtual theory, we will use LMFDB. As explained, LMFDB has several sub-databases - each of which contains different kinds of objects. Within each database, each curve is stored as a single JSON object with common keys[9].

In the following, we will use the elliptic curve database as an example. An example of how an elliptic curve is represented inside of LMFDB can be found in Figure 11.

It can be seen that this entry is a record, i.e. a mapping from keys (in this case strings) to values (any kind of JSON values). This representation encodes for some kind of mathematical object, for example in the case of `isogeny_matrix` this is a matrix – displayed as a list of lists. In the `x-coordinates_of_integral_points` field, this is a little bit more complicated, we are representing a list of integers as a list of strings.

For each entry in LMFDB one OMDoc/MMT declaration has to be generated. To achieve this, the record has to be represented inside of MMT. This in turn requires each JSON value for each key of these records to be translated into an appropriate MMT term, understandable by MMT.

4.2 Translating between Physical and Semantic Representations

To understand how this is achieved, we need to make a difference between the meaning of the values in the database, and how they are represented. We refer to the meaningful, knowledge carrying, version of these objects as their *semantic representation* and the database version as the *physical representation*.

Consider, for example the `degree` field from the example above. This represents the degree of a curve and is an integer value, in this case the integer 1. Inside a database like LMFDB, integer values will usually be represented as a JSON number, i.e. an `IEEE754`

[9] JSON [JSON] is a data interchange format that can easily be written and read by humans. The abbreviation stands for JavaScript Object Notation. JSON values are either `null`, `undefined`, booleans, numbers, strings, (non-homogenous) lists of JSON values or JSON objects. JSON objects are key-value pairs, with keys being strings and values being any kind of JSON values.

```
{
    "degree": 1,
    "non-maximal_primes": [5],
    "torsion_structure": ["5"],
    "ainvs": ["0","-1","1","-10","-20"],
    "x-coordinates_of_integral_points": "[5,16]",
    "real_period": 1.26920930427955,
    "min_quad_twist": {"disc": 1,"label": "11a1"},
    "sha_an": 1.0,
    "conductor": 11,
    "iwp0": 7,
    "2adic_gens": [],
    "torsion_primes": [5],
    "signD": -1,
    "tamagawa_product": 5,
    "isogeny_matrix": [[1,5,25],[5,1,5],[25,5,1]],
    "non-surjective_primes": [5],
    "lmfdb_label": "11.a2",
    "2adic_index": 1,
    "equation": "\\( y^2 + y = x^{3} -  x^{2} - 10 x - 20  \\)",
    "label": "11a1",
    "regulator": 1.0,
    "anlist": [0,1,-2,-1,2,1,2,-2,0,-2,-2,1,-2,4,4,-1,-4,-2,4,0,2],
    "iso": "11a",
    "_id": "ObjectId('4f71d4304d47869291435e6e')"
}
```


Figure 11: An elliptic curve, as found within LMFDB. Some key-value pairs are omitted for readability.

64 bit floating point number. Here this is the floating point value 1.0. But IEEE floats are not able to encode all integers – they have a maximum possible value of $2^{53} - 1$ – so what happens when the semantic representation exceeds that?

Obviously, it is no longer possible to represent the value numerically. Because all values in LMFDB have to be some JSON value, one could encode the integer into a JSON string by making use of the decimal expansion. This would enable storing much larger numbers.

This is where Codecs come into play. Codecs are mappings between the semantic representation (here the integer 1) and the actual representation (the JSON number 1.0). Given a semantic representation of a value it codecs turn it into the corresponding physical representation, and vice-versa.

The set of objects in semantic representation is called the *semantic type*, the set of objects in the physical representation is called the *realized type*. Semantic types reside in the MiTM ontology, whereas realized types resides in the systems themselves. The process of converting between the two representations is called *coding*, specifically coding into a semantic representation is called *encoding*, the reverse is called *decoding*.

4.3 Implementing Codecs In MMT

Codecs			
`codec`	:	`type` → `type`	
`StandardPos`	:	`codec` $\mathbb{Z}^+$	JSON number if small enough, else JSON string of decimal expansion
`StandardNat`	:	`codec` $\mathbb{N}$	
`StandardInt`	:	`codec` $\mathbb{Z}$	
`IntAsArray`	:	`codec` $\mathbb{Z}$	JSON List of Numbers
`IntAsString`	:	`codec` $\mathbb{Z}$	JSON String of decimal expansion
`StandardBool`	:	`codec` $\mathbb{B}$	JSON Booleans
`BoolAsInt`	:	`codec` $\mathbb{B}$	JSON Numbers 0 or 1
`StandardString`	:	`codec` $\mathbb{S}$	JSON Strings

Figure 12: An annotated subset of the `ODK:?Codecs` theory containing a selection of codecs found in MMT. Here $\mathbb{N}$ represents natural numbers (including 0), $\mathbb{Z}$ integers, $\mathbb{Z}^+$ positive integers, $\mathbb{B}$ booleans and $\mathbb{S}$ (unicode character) strings.

The example above corresponds to the `StandardInt` codec, which is commonly used to code integers. A list of codecs along with their realized and semantic types can be found in Figure 12.

In MMT, each codec is present in two ways. First, it exists inside a `Codec` theory inside the foundation part of the MiTM ontology. Each codec is declared along with its' semantic type and corresponds to a single declaration.

As seen in the Figure, this is achieved by first declaring `codec` as taking a `type`, and then declaring each codec with a specific type. This semantic type is represented as a term inside the Math-In-The-Middle ontology.

Second, it is accompanied by a Scala class that implements the translation between semantic and realized type. In the case of LMFDB, this is always a JSON value – so that MMT can understand each value inside each LMFDB record. The concept of codecs however is general and not restricted to JSON objects (or Math-In-The-Middle objects).

4.4 Building Advanced Codecs Using Codec Operators

Building codecs for these simple objects is not enough. Consider for example the `isogeny_matrix` field. The semantic representation of the value of this field is the matrix

$$M = \begin{pmatrix} 1 & 5 & 25 \\ 5 & 1 & 5 \\ 25 & 5 & 1 \end{pmatrix}$$

. In mathematics in general, many objects have a more complex structure, such as tuples, vectors, or matrices (like in this case).

Matrices are characterized three two parameters, the type of object they contain (integers in this case) along their row and column count (3×3 in this case). In principle, one

could construct a codec for each type of matrices by hand. This would mean generating one codec for 1×1 integer matrices, 1×1 real matrices, 1×2 integer matrices, 1×2 real matrices, and so on. For the representation of codecs in MMT, this would require generating one symbol and one Scala function for each different kind of matrix. This quickly becomes a mess.

If one has a codec for integers, one can easily generate a codec for any kind of integer matrices. One can encode each integer as a value and then take this set of representations and store them inside a list of JSON lists. In fact, this is done in the example above and the matrix M is encoded as `[[1.0,5.0,25.0],[5.0,1.0,5.0],[25.0,5.0,1.0]]`.

The procedure of generating codecs is formalized by the concept of codec operators. Essentially codec operators are functors on codecs. These take a codec, along with several parameters, and generate a composite codec. The simple codec operator described here is called `StandardMatrix`.

4.5 Codec Operators in MMT

Codecs (continued)			
`StandardList`	:	$\{T\}$ `codec` $T \to$ `codec` List(T)	JSON list, recursively coding each element of the list
`StandardVector`	:	$\{T, n\}$ `codec` $T \to$ `codec` Vector(n, T)	JSON list of fixed length n
`StandardMatrix`	:	$\{T, n, m\}$ `codec` $T \to$ `codec` Matrix(n, m, T)	JSON list of n lists of length m

Figure 13: Second annotated subset of the `ODK:?Codecs` theory containing a selection of codec operators found in MMT. Compare with Figure 12.

A list of codec operators can be found in Figure 13. Codec Operators in MMT are again represented in two ways, as declarations inside the `Codec` Theory and inside a Scala implementation.

Unlike simple codecs, codec operators do not directly specify a semantic type. As can be seen in the Figure, they take several parameters used for the resulting semantic type. These are shown in curly brackets. For the `StandardMatrix` codec above these are ⟪T⟫ (the type of elements in the matrix), ⟪n⟫ and ⟪m⟫ (row and column counts). The codec then also takes a codec for ⟪T⟫, to then finally return a codec for a composite semantic type.

As each operator is an MMT declaration, with appropriate type, MMT terms can be used to represent codecs created by codec operators. Consider now the codec used to encode the 3×3 integer matrix M above. This corresponds to the MMT term `StandardMatrix`$(3, 3,$ `StandardInt`$)$ [10]. Similarly the same codec operator can be used to

[10] The observant reader will have noticed that the way codec operators have been declared, the codec in question actually corresponds to the term `StandardMatrix`$(\mathbb{Z}, 3, 3,$ `StandardInt`$)$. This has an additional $\mathbb{Z}$ as the first argument. However, the last argument is a codec for a specific semantic type and thus fully

for example generate a codec for 2×2 boolean matrices, which corresponds to $\texttt{StandardMatrix}(2, 2, \texttt{StandardBool})$.

4.6 Declaring Schemas to Translate Records and Implement virtual theories

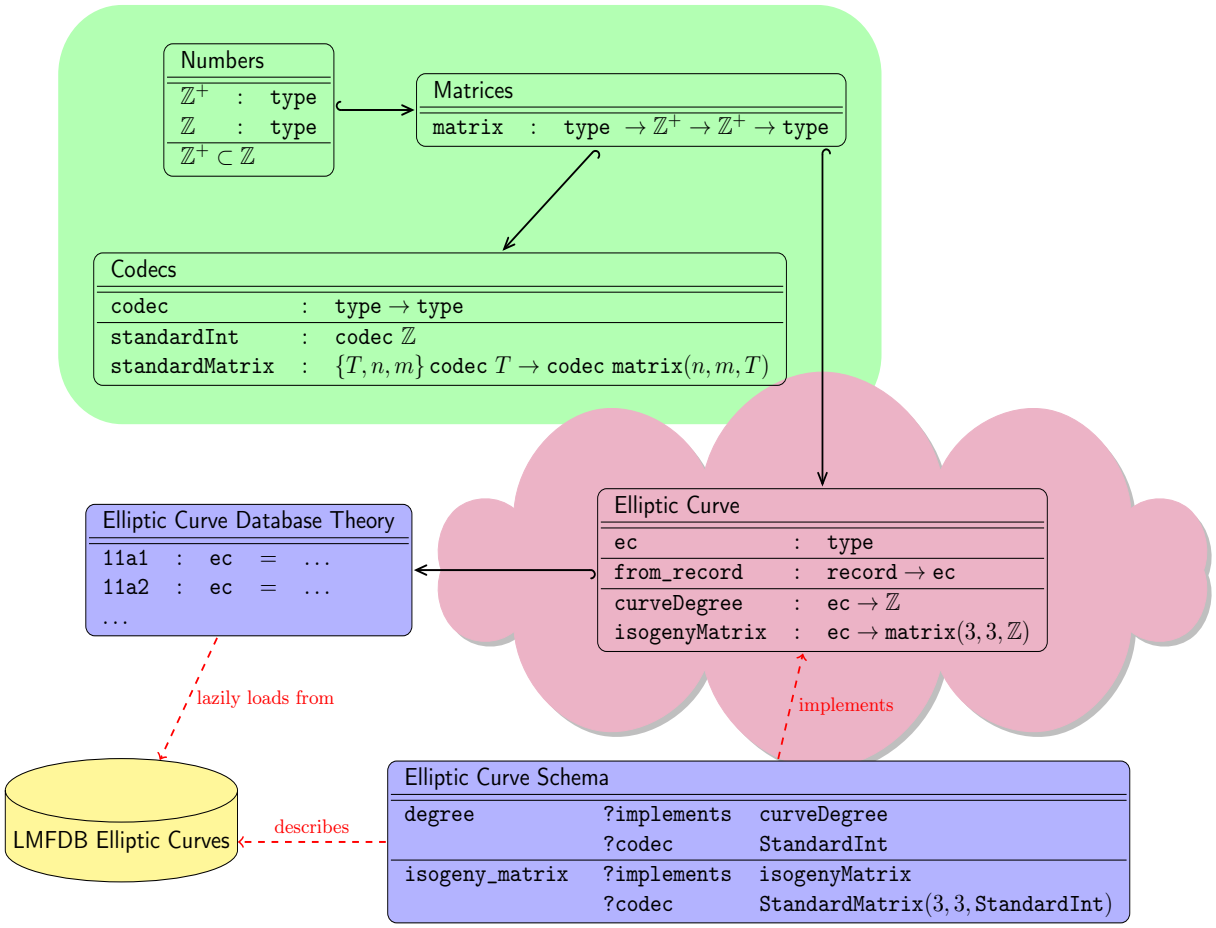


Figure 14: A sketch of the architecture for a virtual theory connecting to LMFDB. Solid edges represent imports. Several declarations have been omitted for simplicity.

Codecs enable creation of individual values within LMFDB and mathematical databases in general. This is not enough – a mechanism to translate entire records as a whole is needed to solve aspect (i).

The architecture of a virtual theory for LMFDB elliptic curves is illustrated in Figure 14. The figure is split into four major parts, colored analogous to the ontology parts in Figure 8.

determines the first argument. MMT is capable of transparently inferring the first argument, thus it can be omitted for readability without needing any kind of special treatment implementation wise.

These parts are the foundational ontology theories (green), mathematical model ontology (red), database interface theories (blue) and LMFDB itself (yellow).

Recall that the foundational ontology theories provide a system-independent basis for the remainder of the approach. In this example, they first define a type of integers $\mathbb{Z}$ and positive integers $\mathbb{Z}^+$ and then proceed to define a `matrix` type. This type takes three parameters, a type of elements in the matrix, and then a row and column count. Next, the codec `standardInt` and codec operator `standardMatrix` are defined as previously.

Next, the *Elliptic Curve* theory is described. It is contained in the mathematical models part of the ontology. It models an elliptic curve in a very simple fashion, by just declaring a type `ec`. Next, it defines a `from_record` constructor that takes an MMT record and returns an elliptic curve. Notice that these definitions are independent of the LMFDB database.

The theory then moves on to define the two important properties of elliptic curves. These are *degree*, an integer, and the *isogeny matrix*, an integer matrix of size 3×3. In reality there are of course more than these two – the others can be implemented analogously and are omitted here to better illustrate this example.

The two properties are modeled as functions that take an elliptic curve and return the appropriate type. Recall that the Math-In-The-Middle approach aims to model mathematical knowledge "in the middle" independent of any particular system. This is exactly the case here – the model of elliptic curves does not rely on LMFDB, nor any other system. As explained, this choice was made deliberately so that this Virtual Theory approach can eventually be extended to include other systems or can be easily updated, should the structure of LMFDB be changed.

Notice however, how the properties make use of integers and matrices. Thus the elliptic curve theory includes the Matrices and Numbers theories, making use of the foundations.

Next, we investigate the interface theories that interact with LMFDB. The mechanism here is flexible to be extensible to other systems. It consists of two theories, the so-called *schema theory* and the *database theory*.

The schema theory, as the name suggests, describes the schema of the LMFDB elliptic curve database. This is the only place in the entire architecture of virtual theories which relies on the structure of LMFDB. The schema theory has the URI `lmfdb:schema/elliptic_curves?curves` and contains declarations for each field within an LMFDB record. The name of these declarations corresponds to the name of the field inside the record. Each declaration is annotated using MMT meta-data with two pieces of information, the property of an elliptic curve it implements and the codec that is used to encode it inside LMFDB. For example, the `degree` field implements the `curveDegree` property in the elliptic curve theory and uses the `StandardInt` codec.

Next comes the database theory. This is the truly virtual theory – it is not stored on disk and generated dynamically – and has the URI `lmfdb:db/elliptic_curves?curves`. It contains one declaration per record in LMFDB.

4.7 Translation of the 11a1 Curve

But how is this database theory generated? It uses a so-called `Backend` – an MMT abstraction used to load declarations into memory. Upon being given a URI, it is responsible for loading the underlying definition. For the elliptic curve theories these URIs are of the form `lmfdb:db/transitivegroups?groups?11A1`.

The backend first retrieves the appropriate record from LMFDB – in the case of `11A1` this corresponds to retrieving the JSON found in Figure 11. Next, the backend attempts to turn this JSON into an MMT record so that it can be passed to the `from_record` constructor.

For this, it needs all declarations in the schema theory. Each of these declarations corresponds to a single field in the JSON, that can be turned into a field of the MMT record. In the example provided here, we only consider two fields, `degree` and `isogeny_matrix`.

For each of these two fields, the backend knows which field to create in the MMT record that it has to construct. They are given by the `?implements` meta-datum, here `curveDegree` and `isogenyMatrix`. But this information is not enough. The JSON values of the fields can not be used as values inside an MMT record, they need to be assigned their correct semantics first.

This is where codecs and the `?codec` meta-datum come into play. The physical representation of the `degree` field is 1, a JSON integer. The schema theory says that this is encoded using the `StandardInt` codec from above. To generate an MMT value for the record, this codec can be used to decode it. In this case the decoded value is the integer 1[11].

The physical representation of `isogenyMatrix` is `[[1.0,5.0,25.0],[5.0,1.0,5.0],[25.0,5.0,1.0]]`. Here, the schema theory contains a codec that is constructed using the `StandardMatrix` codec operator, specifically `StandardMatrix`$(3, 3,$ `StandardInt`$)$. To apply this codec, the Backend has to first construct the concrete codec, which can then used to decode the physical representation. Since this is a codec operator, first each entry of the matrix has to be decoded using `StandardInt` – turning the JSON number 1.0 into the integer 1, the JSON Number 5.0 into the number 5, etc. Then these decoded values can be placed inside a matrix to arrive at the semantic representation[12]

$$M = \begin{pmatrix} 1 & 5 & 25 \\ 5 & 1 & 5 \\ 25 & 5 & 1 \end{pmatrix}$$

[11] In this document, the physical and semantic representation are rendered in the same fashion. It is important to realize that they are not in fact the same. The physical representation is a 64-bit floating point JSON Number 1, whereas the semantic representation is the integer 1. Technically, the semantic representation is actually the OMDoc/MMT integer literal 1. We skim over this detail here, as the OMDoc/MMT literals are designed to precisely represent this value.

[12] Similar to the semantic representation above, the matrix M is technically different from the OMDoc/MMT representation. We could again represent this using a matrix literal, but instead the implementation actually uses a constructor containing integer literals. For simplicity, and as literals are designed to precisely represent mathematical objects, we omit this detail.

.

This gives the Backend all the information it needs to construct an MMT record which can then be turned into an elliptic curve using the `from_record` constructor. The `degree` field is assigned the value 1 and the `isogenyMatrix` is assigned the value of the matrix M. Finally, this MMT term can be used to define a new constant inside the database theory.

This example only shows the translation of two fields. But this can obviously be extended to more than these two – for example the other ones contained in the JSON received from LMFDB. One only needs to add the appropriate declaration in the schema theory.

Furthermore, this can also be extended to more than a single LMFDB sub-database, by adding appropriate new Schema and Database theories along with mathematical models. Recall that aspect (i) of the solution of the research question requires a generic, extensible mechanism to manage knowledge stored in different systems. This is exactly achieved here, by separating physical representation (Codecs), database structure (Schema Theory) and Semantics (Elliptic Curve Theory in the Math of Math-In-The-Middle).

5 The QMT Query Language

Now that we have seen how to represent and maintain knowledge in the MiTM ontology, we can turn towards aspect (ii) of the research question. There is a multitude of problems where is is not enough to give the URI of a specific declaration and then retrieve this via MMT. There are several queries where one either does not know the exact URIs of all items to be retrieved, or where one wants to find all declarations subject to a specific criterion.

To allow for more flexibility and enable general "information retrieval" MMT has a query language. This query language is called QMT – Query language for Mathematical Theories. QMT started as a query language for formal libraries [Rab12], but within the scope of this project it has been extended to better fit our needs.

The query language enables the user to find a set of objects – this can be MMT objects (i.e. terms), declarations or even a set of theories and views – subject to specific criteria.

Example 15. One simple use case is to find all declarations inside of a given theory – for example the query

```
Related(
       Literal(URI('odk:algebra?Semigroup')),
       ToObject(Declares())
)
```

when evaluated returns the three URIs to $G, \circ$ and `assoc` from Figure 4.

Understanding even this simple query is non-trivial. We leave it to the below once we have gone over some parts of the query language to reiterate this example.

5.1 The Inductive Nature of the Query Language

The first important thing to note is that the query language is an inductive data structure. This decision was made for two reasons, (1) to enable typing of the language and (2) to allow easier implementation of evaluation. Type-checking of the language can prove very useful in practice, for instance to check if one has formulated a query correctly. It is also used during evaluation – but more on this later.

Being inductive in nature, the query language contains certain basic queries, that take few to no parameters, and complex queries, that take queries themselves as parameters. The query language is able to express information about different forms of content inside of MMT. It can return URIs, Objects and String literals – we refer to these as `QueryBaseType`s. Example 15 above is asking for a set of URIs – namely all the URIs that are declared within the theory `odk:algebra?Semigroup`.

Different queries may return different types of results. Specifically, a query can return either a typed tuple of `QueryBaseType`sÂă or a homogeneous set of such tuples [13].

[13] In this setting "typed tuple" refers to the exact typing of each element in the tuple. For example, we consider a tuple of type `(URI, Object)` as typed but not one of type `(QueryBaseType, QueryBaseType)`.

Example 16. Another question one might ask is, "what are the divisors of elliptic curves inside of LMFDB that have a conductor divisible by 5". In fact, this question came up during the OpenDreamKit project when asking John Cremona about what queries the existing LMFDB architecture could not handle[14].

Such a query can be expressed inside the query language – although it gets very complex very quickly – thus we omit it here. Unlike the previous example, this one returns a set of MMT objects – namely those representing divisors.

The grammar of the query language consists of several parts: `Queries`, `Propositions`, `Relations`, `Predicates` and `Judgements`. An overview of each of these is given in the following, in particular detailing which queries these can be used to answer.

Inside the query language, the `Query` class represents evaluable queries. The structure of queries is very much related to evaluation and intentionally kept restricted in certain situations. They can be roughly classified into three different types:

1. Terminal Queries, which do not have sub-queries as parameters,
2. Complex queries, formed by query operators which that take a sub-query as a parameter and upon evaluation evaluate the subquery and use the result thereof, and
3. Technical Queries, which have no direct influence on the result but only on how the query is evaluated

The aim of this section is to give the reader an overall understanding of the query language, thus some cases are omitted. The interested reader is referred to the MMT website [MMTa] and in particular the source code for a complete list. Furthermore, the following talks about evaluating queries and at the same time treats them as functions that have a return value – both of these refer to the same thing.

Terminal Queries and Unary Predicates

The first type of terminal queries are those that return object literals – appropriately called `Literal(`$\langle\!\langle obj \rangle\!\rangle$`)` and `Literals(`$\langle\!\langle objs \rangle\!\rangle$`)`. For example, a query `Literal(URI('odk:algebra ?Semigroup'))` directly evaluates to the URI literal `URI('odk:algebra?Semigroup')`.

Unary predicates effectively function as types of MMT URIs – they state which kind of objects the URIs refer to. For example, the unary predicate `IsTheory()` applies to URIs like `odk:algebra?Semigroup` and `odk:algebra?Algebra` – all URIs that contain exactly one "?" and thus point to theories. Other important unary predicates include include `IsView()` and `IsConstant()`.

Unary predicates form the basis of the second type of terminal queries – the `Paths(`$\langle\!\langle un \rangle\!\rangle$`)` query.

Example 17. The query `Paths(IsTheory())` returns the paths to all theories known to MMT.

[14] It turns out that this query could actually be handled by existing architecture. However, this was not an intended feature and a side-effect of an arbitrary code execution vulnerability on the side of the server.

In general, path queries return the set of paths that are subject to the given unary predicate.

As queries are inductive in nature, these two terminal queries form the basis of all queries. This choice was made intentionally – it enables a functional implementation of the query language to be contained in a certain sense. A functional implementation has to evaluate each sub-query individually and at one point have each of the intermediate results in memory. This choice makes sure that the sets each implementation has to work with are limited – one never has to work with all paths in MMT at once.

5.2 Query Results as Sets

The second type of queries are *Complex Queries* made up of *Query Operators*. These take queries as parameters and return a new query.

Example 18. The query

```
Union(
  Paths(IsTheory()),
  Paths(IsView())
)
```

returns the set of all paths that are either theories or views.

This example demonstrates an operator that treats query results as sets – simply lifting the union operator to the query level. These types of queries do not depend on the semantics of content in MMT. Apart from the `Union(`⟪*left*⟫`,`⟪*right*⟫`)` operator, complex queries can also be formed by `Intersection(`⟪*left*⟫`,`⟪*right*⟫`)` and `Difference(`⟪*left*⟫`,`⟪*right*⟫.

Making Use of Binary Predicates

On top of unary predicates of MMT paths there are also binary predicates. As binary predicates do these relate two MMT URIs to each other – for example the binary predicate `Declares` relates the subject `odk:algebra?Semigroup` to the object `odk:algebra?Semigroup?G`. Other binary predicates include `Declares`, `HasDomain`, `HasCodomain` and `Includes`.

To be able to use these predicates inside of queries the `Related(`⟪*to*⟫`,`⟪*by*⟫`)` operator exists. We have already seen an example for this, Example 15 above was the first example introduced. This demonstrates the use of this operator – the ⟪*to*⟫ parameter generates a set of URIs and is then filtered by everything related by a given relation ⟪*by*⟫.

It should be noted that a single binary predicate can not directly function as a relation. To elaborate this further, consider a query that returns the single URI `odk:algebra?Semigroup` and set it as the ⟪*to*⟫ parameter for a `Related` query. Looking into the `Declares` relation one has two choices – find all URIs which are declared by this theory or find all URIs which declare this theory. `ToSubject` and `ToObject` can be used to clarify which one is requested.

Binding Variables in Queries

Another type of query operators allows for binding of variables. These created named references to intermediate results of the query and have undergone a refactoring and expansion during this project. The most relevant query operator is the `Comprehension(`⟪*domain*⟫`,` ⟪*varname*⟫`,`⟪*pred*⟫`)` – which allows to filter a specific domain by those that fulfill a specific predicate.

Example 19. The query

```
Comprehension(
       Related(
              URI('odk:algebra?Semigroup'),
              ToObject(Declares())
       ),
       LocalName('x'),
       Holds(
              Bound(x),
              Typing('x', OMV(LocalName('x')), OMID(odk:algebra?Semigroup?G))
       )
)
```

finds all declarations within the *Semigroup* theory that have a type 'G'.

We start on the left hand side by finding all URIs to declarations within the *Semigroup* theory. This is the same as in Example 15. Next, we use the comprehension operator to find all of those xs which are subject to a specific predicate. In general, `pred` parameter refers to a predicate, which in the codebase is represented by a class on its own.

Here we specifically make use of the `Holds(`⟪*about*⟫`,`⟪*j*⟫`)` predicate, which has been added to the grammar language to allow interaction with the object level of MMT. It can check if a so-called `Judgement` holds for the current variable (x). We declare a local variable within the query called x inside the Comprehension operator. We then refer to it within the predicate by using using `Bound(x)`. If we did not allow for binding of variables, we would only be able to declare predicates that are universally true or false – resulting in the Comprehension operator becoming useless.

Other kinds of queries that allow binding include

- the `Let(`⟪*varname*⟫`,`⟪*value*⟫`,`⟪*in*⟫`)`, which explicitly make a variable available,
- the `BigUnion(`⟪*domain*⟫`,`⟪*varname*⟫`,`⟪*of*⟫`)` operator, and
- the `Map(`⟪*varname*⟫`,`⟪*in*⟫`,`⟪*term*⟫`)`, which constructs a new term for each element of the original query.

The last `Map` query has been added during this thesis to make certain computational tasks easier to express.

Judgments, as given to the predicate by the j parameter, are represented by the `QueryJudgment` class. This is a new addition to QMT. It allows users to dynamically look into the MMT types and structure of objects and check if they are subject to certain

conditions. In this specific example we use the `Typing` Judgment to check if a variable has a type G.

5.3 Evaluating Queries of MMT Content

Next, the reader is provided with an overview of how queries are evaluated. This exploits the typing of the query language, in particular return value of the query can be predicted without having to evaluate the query.

Recall that MMT queries can return either a typed tuple of `QueryBaseType`s or a list thereof. This allows to have a query always return a set of tuples of `QueryBaseType` internally. A single tuple `QueryBaseType` can be represented by an appropriate singleton – while separately maintaining the right typing for the results.

We use the inductive nature of queries to guide us in the evaluation. Here a single query is considered as a (syntactic) tree, and the leaves are used start an inductive evaluation procedure. During this procedure, a big initial result set is generated for the terminal queries, and then used inductively to make this set smaller. As mentioned above the terminal queries are intentionally limited – to allow us to keep all of these intermediate results in memory.

Recall that two important terminal queries are the `Literal` and `Paths` queries. The first query of these is trivial to evaluate – simply place the literal inside of the query into the intermediate result set – whereas the second one is a bit more tricky.

Evaluation of Predicates and Relations

To evaluate a `Paths` query – like the one in Example 17 – all URIs that are subject to a given unary predicate need to be found Assuming that all content is stored on disk, this can be solved in an elegant fashion.

Whenever new content is added to MMT, it is indexed and all information about which objects are of which type are added to an index. This index comes in the form of a `RelationalStore`, which is managed by a `RelationalManager` (the naming will become more obvious later). At evaluation time, the implementation can easily look into this index and quickly retrieve all URIs of a specific type.

Next, the evaluation of query operators is explained. Only two of four cases are non-trival.

In the case where these operators simply lift set operations to operations on queries, the evaluationÂăis straightforward – the set operations are just performed on the intermediate results. Now for the interesting cases, we have a look at the `Related(`$\langle\!\langle to \rangle\!\rangle$`,`$\langle\!\langle by \rangle\!\rangle$`)` operator. It has to be aware of binary predicates. The implementation makes use of the same trick as above with the unary predicates – whenever new content is added, all possible binary predicates are cached and store in an index. This again happens inside of the `RelationalStore`. Now the name makes sense as it stores relational information about a given set of knowledge items. During evaluation, the index can be used to quickly retrieve all items related to a given URI with a given predicate.

Thus the query in Example 15 is evaluated in two steps. First, one evaluates the argument `Literal(URI('odk:algebra?Semigroup'))` – it is a literal query so it just returns the argument. Then one looks into the `RelationalStore` and finds all stored binary predicate triples, where the relation is `Declares` and the object is the URI `odk:algebra?Semigroup`. This will give exactly what we the query is looking for – the URIs to constants declared by the *Semigroup* theory from Figure 4.

5.4 Evaluation of Predicates Involving Bound Variables

Next, the evaluation of the `Comprehension(`⟪*domain*⟫`,`⟪*varname*⟫`,`⟪*pred*⟫`)` operator is described. For this, recall Example 19. Like with the other query operator the argument needs to be evaluated first. This is the exact query given by Example 15 and returns a set of URIs to concepts.

Next, an iteration is required. For each element of this intermediate result set the other important argument to the query – a predicate – needs to be evaluated. Specifically, it needs to be checked if the

```
Holds(
       Bound(x),
       Typing('x', OMV(LocalName('x')), OMID(odk:algebra?Semigroup?G))
)
```

predicate applies or not.

For this purpose a sub-query is spawned multiple times – one for each of the elements inside the original set. These sub-queries evaluate the predicate and need to know which element of the intermediate results the x stands for. Thus it needs to be given a context. In this case, a context is simply a set of mappings for variables.

The QueryJudgement

```
Typing('x', OMV(LocalName('x')), OMID(odk:algebra?Semigroup?G))
```

needs to be evaluated. Principally, this can be achieved by a part of MMT called the Solver.

The actual evaluation is rather technical and first a so-called `Stack` for the Judgment is needed. The Stack contains a set of rules that can be used to evaluate the judgment. To hide the process of having to retrieve this Stack first from the User, the QueryJudgment class has been introduced. This automatically retrieves all needed information at query evaluation time and passes on the information if the judgment holds or not from the Solver.

After the evaluation of sub-queries has been finished, is is easy to generate the result set. Only the set of elements for which the predicate applies – which are known at this point – should be returned. Other queries that bind variables make use of a similar procedure, with the exception that they do not check if a predicate applies, but instead evaluate different subquery results.

6 Cross-System Communication Using Queries

The Math-In-The-Middle approach and the implementation including virtual theories has been discussed. QMT, a query language for MMT has also been described. This section focuses on how to make use of these concepts to enable transparent system interaction.

So far, QMT has only been used on concrete MMT content, and in particular the evaluation has made use of indexes. These indexes are built incrementally – each time new content is added to MMT, it is added to the index. Content can only be found using QMT if it is stored in the index.

6.1 Avoiding Reliance on In-Memory Content

Recall that virtual theories only load content lazily – exactly when it is required. This means that an item is only added to the index – and can only be found by QMT queries – once it is explicitly requested by the user. Thus if the user wants to properly resolve a query, they first have to load all potential results into MMTmemory. This is best illustrated with an example.

Example 20. The query

```
Related(
       URI('lmfdb:db/transitivegroups?groups'),
       ToObject(Declares())
)
```

when evaluated should return the URI of all known transitive groups.

To evaluate this query, the existing implementation would first look into the index for binary predicates to find all URIs that are related to the transitive group theory by the given `Declares()` relation. Here is where the problem comes in – where does this index come from?

In order to build such an index, (at the very least) a complete list of available groups is required. By nature of being a virtual theory, such a list never exists in MMT memory by default – since the backend for the transitive groups theory only loads the set of declarations that is needed. Thus such an index is at most partial – and may not exist at all.

The same not only goes for this specific QMT query – it holds for all queries sent to virtual theories. As seen, evaluation can break easily and may only return partial results, if any at all. This limitation makes search fairly useless for virtual theories. Furthermore, it also negates the advantage of virtual theories. These were built so that not all content has to be loaded from an external system into memory.

Thus the evaluation of queries had to be reworked. There are several approaches to achieve this.

One obvious approach involves loading all relational information into memory. Once this is available, the same strategies as for concrete theories can be used. This still poses a similar problem to the one that virtual theories are trying to solve in the first place. Such an approach still needs to load information about the entire external database into

the MMT system – albeit to a lesser degree because only need the relational information. Thus this solution is not a desirable one.

6.2 Making Use of Existing Querying Mechanisms

Most external systems already have some form of information retrieval mechanism – commonly in the form of a query language or an API. And indeed, in the case of LMFDB, such an API exists.

One can send queries to the underlying MongoDB implementation. Answering Example 20 seems almost trivial – a simple `GET` request to `http://www.lmfdb.org/api/transitivegroups/groups/?_offset=0&_format=json` is enough. The LMFDB API is also capable of answering more interesting queries.

This provides a new approach for making queries towards virtual theories. First, the MMT query is translated into a system-specific information-retrieval language – in the case of LMFDB this is a MongoDB-based syntax. Next, this translated query is sent to the external API. Upon receiving the results, these are translated back into OMDoc/MMT with the help of already existing functionality in the appropriate virtual theory backend.

This leaves just one problem unsolved – translating queries into the system-specific API. Consider that it is not sufficient to just translate all queries. One hand a general QMT query may or may not involve a virtual theory. On the other hand, it may also involve several unrelated virtual theories. This makes it necessary to filter out queries involving virtual theories, so that they can be evaluated properly.

Achieving this automatically is a non-trivial problem. Queries are inductive in nature, along with their evaluation. In principle, one could intercept each of the intermediate results. However, this would require a check on each intermediate result to determine if it comes from a virtual theory or not,

Recall that queries may return URIs, Objects and String literals. Inspecting URIs is rather simple, but this is not the case for objects. The objects themselves are inductive in structure and thus recursing through the entire structure would become necessary.

Thus such an approach would be very expensive computation wise and would not scale if a query returns a large set of objects. This would not only cause a performance hit, but also make for a much more complicated codebase. After each intermediate result one potentially has to switch the entire evaluation strategy.

6.3 Annotating Sub-Queries for External Systems

Instead, during this project a new type of query was introduced – the `I(`$\langle\!\langle query \rangle\!\rangle$`,`$\langle\!\langle hint \rangle\!\rangle$`)` operator. The first parameter of this is a query that is to be evaluated, and the second one is a so-called evaluation hint that describes how the query should be evaluated. This second parameter currently is a simple string.

The goal of this type of query is to annotate a sub-query and tell the implementation to use a specific external theory to evaluate it. Each virtual theory backend can register

a `QueryEvaluationExtension`. This is responsible for translating these sub-queries and retrieving the appropriate results.

With this new operator, Example 20 becomes:

Example 21. Example 20 annotated with the `I()` operator.

```
I(
	Related(
		URI('lmfdb:db/transitivegroups?groups'),
		ToObject(Declares())
	),
	'lmfdb'
)
```

Evaluation of such a query is straightforward. If this query is encountered during query evaluation, the implementation finds the `QueryEvaluationExtension` identified by the hint. Then it turns over control to the extension and allows it to evaluate results. Once the results are available, control is passed back to the normal QMT evaluator which continues evaluating as normal. Here, this query is used for virtual theories, however it could in principal be used to implement other evaluation strategies as well.

This procedure has turned out to work very well in practice. However, this does not fully address our requirement. The user needs to explicitly mark up virtual theories and thus they are not as transparent as originally desired.

6.4 Calling the LMFDB API

It is not easily possible to translate all queries into LMFDB Queries. The purpose of this subsection is to give the reader an insight into which queries can be translated and how this is achieved.

To identify this subset of translatable queries, we look into the structure of knowledge stored in LMFDB. Recall from Section 4.1 that each item is essentially a set of key-value pairs. As such, interesting queries usually take the form of "find all objects in this LMFDB database where some specific key equals a specific value".

Example 22. The query

```
I(
	Comprehension(
		Related(
			URI('lmfdb:db/transitivegroups?group'),
			ToObject(Declares())
		),
		LocalName('x'),
		Holds(
			Bound(x),
			Equals(
				'x',
				OMV(LocalName('x')),
				OMA(
```

```
                                        OMID('mitm:smglom/algebra?magma?commutative'),
                                        OMV(LocalName('x'))
                                ),
                                OMLIT(mitm:Foundation?Logic?bool?true)
                        )
                )
        )
        , 'lmfdb'
)
```

finds all commutative transitive groups known to LMFDB.

This example is typical for queries that can be translated into LMFDB. It begins on the outside with the `I` operator, to indicate that this is a query to be evaluated using LMFDB. Next comes a composite query built using the `Comprehension` operator. This operator is applied to two main arguments, (1) the set of all transitive groups (recall example 20 above) and (2) a predicate that checks if a group is commutative.

To translate this QMT query into an LMFDB query, two main steps need to be performed. First, the sub-database of LMFDB that is involved needs to be found. This can be determined from the first argument to the Comprehension operator, here this is the `transitivegroups` database. Second, the groups to select from the database need to be selected.

This second step is the more restrictive one. We need to have a closer look at the `Equals` judgment

```
Equals(
        'x',
        OMV(LocalName('x')),
        OMA(
                OMID('mitm:smglom/algebra?magma?commutative'),
                OMV(LocalName('x'))
        ),
        OMLIT(mitm:Foundation?Logic?bool?true)
)
```

.

This again has several arguments,

- a left hand side and a right hand side of the equality to be checked and
- a variable (here x) that needs to be substituted.

The left hand side of the equality is given by the application of `mitm:smglom/algebra?magma?commutative` onto this variable. The right hand side is given by a literal value of true. Thus it can be inferred that LMFDB should return all groups for which the key corresponding to the `commutative` property has the literal value `true`.

Next, one can make use of codecs to find the name of the field corresponding to the `commutative` property. This is the `ab` (as in abelian) field. Furthermore, the codecs can be used to translate the literal `true` into a JSON value understood by the LMFDB

API. Here, the codec `BooleanAsInt` is used and the value of `true` corresponds to 1. This generates an API call to `http://www.lmfdb.org/api/transitivegroups/groups/?ab=i1&_offset=0&_format=json`.

6.5 QMT Queries as OpenMath Objects

Next, an introduction on how to send queries to MMTis given. Previously, to send a query to MMT, it was either required to directly interface with MMT on a Scala level or to encode queries in an XML representation. The first approach is infeasible to most system as they can not access Scala objects natively. The second approach provides an MMT specific XML syntax, that might not be easily adaptable by other systems.

The SCSCP protocol described Section 3.4 is designed to enable exactly the kind of interoperability that is needed here. It allows different mathematical knowledge systems to communicate using a common language, namely OpenMath objects. Thus it makes sense to represent queries as OpenMath objects, so that they can be communicated using this protocol.

Recall that the QMT grammar is inductive. This enables representing each query as a term in MMT.

One can start by representing each terminal query and each query operator as an MMT symbol. Since the `Query` class recuses into `Prop`, `Unary`, `Binary`, `QueryJudgement` and other classes, it needs to be done for these as well. This gives a set of theories representing each class. These can be found in the namespace given by the `qmt` abbreviation.

This can be easily used to represent terminal queries, by using an appropriate reference to the symbol. To represent a complex query, an application of the query operator symbol to the terms representing the query parameters. Furthermore, literals can be used for queries with integer and string parameters. Note that the `QueryJudgement` class recurses into the `Term` class which can be used directly – no special symbols needed.

Furthermore, special care needs to be taken of binding queries. Here a single application is not enough. As the name says, the queries are binding variables. Hence we can make use of a so-called binder term, an OpenMath term that allows the binding of variables.

Recall that MMT terms correspond to OpenMath objects. This means that if one can represent `Queries` as MMT terms, one can also be represent them as OpenMath objects. In Examples 23 and 24 we can see the OpenMath objects[15] that correspond to the Queries in Examples 15 and 22 respectively.

Example 23. Example 15 as an OpenMath object.

```
OMA(
  OMS(URI('qmt:?QMTQuery?Related')),
  OMA(
    OMS(URI('qmt:?QMTQuery?Literal')),
    OMS('odk:algebra?Semigroup')
  ),
```

[15] The syntax used in these examples is not valid OpenMath. This would usually be represented using XML, however to make it easier on the reader an MMT surface syntax like style is used.

```
  OMA(
    OMS(URI('qmt:?QMTRelationExp?ToObject')),
    OMS(URI('qmt:?QMTBinaries?Declares'))
  )
)
```

Example 24. Example 22 as an OpenMath object.

```
OMA(
  OMID(URI('qmt:?QMTQuery?I')),
  OMBIND(
    OMID(URI('qmt:?QMTQuery?Comprehension'))
    OMV(LocalName('x')),
    OMA(
      OMS(URI('qmt:?QMTQuery?Related')),
      OMA(
        OMS(URI('qmt:?QMTQuery?Literal')),
        OMS('lmfdb:db/transitivegroups?group')
      ),
      OMA(
        OMS(URI('qmt:?QMTRelationExp?ToObject')),
        OMS(URI('qmt:?QMTBinaries?Declares'))
      )
    ),
    OMA(
      OMID('Holds')
      OMV(LocalName('x')),
      OMBIND(
        OMID(URI('qmt:?QMTJudgements?Equals'))
        OMV(LocalName('x')),
        OMA(
          OMID('mitm:smglom/algebra?magma?commutative'),
          OMV(LocalName('x'))
        ),
        OMLIT(mitm:Foundation?Logic?bool?true)
      )
    )
  ),
  OMSTR('lmfdb')
)
```

6.6 Using MMT Surface Syntax

The above examples demonstrate that the OpenMath syntax of queries almost directly corresponds to the inductive structure of QMT. This OpenMath approach easily allows computer systems to formulate queries, however it is equally complex to create queries in for a human. To make MMT queries accessible for humans, it is necessary to design a more human-readable syntax.

Recall that MMT terms can be parsed from the human-writable MMT surface syntax. With the help of notations, syntax for queries that can be both read and written by end-users can be defined. When designing such a syntax to be parsed by MMT, two factors need to be taken into account.

First, each query usually contains a multitude of query operators and references to content within MMT. This poses a problem because both need to refer the symbols representing the query operators and the actual symbols somewhere inside of MMT as symbols.

Example 25. Example 15 as an MMT object in surface syntax.

```
related to ( literal ‘odk:algebra?Semigroup ) by (object declares)
```

An example makes this problem more obvious. Consider Example 25, specifically the inner part `literal ‘odk:algebra?Semigroup` only. This corresponds to a `Literal` query with an argument of `odk:algebra?Semigroup`. One needs to make sure that the second argument to this query is parsed as a literal of a given URL, and not as part of the query syntax. To achieve this, one uses a generic feature in the MMT parser and simply mark it as a literal by prefixing it with `‘`.

Example 26. Example 22 as an MMT object in surface syntax.

```
use "lmfdb" for {*
  x in (
    related to ( literal ‘lmfdb:db/transitivegroups?group ) by (object declares)
  ) | holds x (
    x commutative x *=* true
  )
*}
```

The second and related problem comes when using MMT terms within parts of a query. Consider Example 26. Here, inside the `holds` part of the query, one wants to find all groups inside of LMFDB that are commuatative. This is written as `commutative x *=* true`. Here it is not that difficult to separate the actual terms from the query level, but in a general scenario, there could be an arbitrary term instead. This could contain any kind of references to any kind of symbols inside of MMT.

To solve this problem, one has to make sure to always find all query level operators first, then construct an MMT term representing the query and finally parse the internal MMT terms. This is achieved by giving the Query operators a lower precedence in parsing. Furthermore, to make sure that all sub-terms are parsed correctly, one makes sure that MMT knows the theory these literal terms come from, by adding an appropriate reference to the context the entire syntax is parsed in.

6.7 Sending Queries from the Web

We discuss how to make use of the QMT query language. We have looked at QMT and seen how it can be used to enable transparent cross-system communication and distributed

computation. Furthermore, we have seen how it can be given both a human readable and machine readable syntax.

The next step is to expose an interface for queries to be sent to the MMT system and be evaluated.

The first thing we want to do is build an API with which other systems can send QMT Queries to MMT and receive responses. A common way for almost any kind of software systems to communicate is via an HTTP based API - a RESTful interface. This is easy to implement in almost any kind of programming language and thus enables a broad set of clients to communicate with MMT efficiently.

MMT already has a built-in webserver and also an extension mechanism that allows plugins to add custom functionality to it. Prior to this project, a minimal plugin that allowed to send queries in a legacy XML-based syntax existed. This plugin has been refactored and extended to support surface syntax, as shown in Section 6.6.

Upon receiving a query from a client, this plugin first parses the syntax and converts it into the internal data structure representing the query, before type checking it to ensure validity. Next, the plugin evaluates the query using the procedures described in Sections 5.3 and 6.4. Finally, the result is serialized into XML and sent back to the client.

To demonstrate this interface and make it accessible to users a web interface has also been implemented. Screenshots of the interface can be found in Figures 27 and 28.

Figure 27: QMT Web Interface for entering Queries.

The web interface consists of two parts, an interface to enter queries as well as a result listing.

The interface for entering queries can be seen in Figure 27. To submit a query, the user has to fill out two fields and click the `Query` button. The fields to be filled are the context and the query itself.

As described previously, the context is a set of theories with respect to which the queries should be parsed. It can be entered in the `Context` text box separated by commas. The query, in surface syntax, can be entered in the main text area on the left.

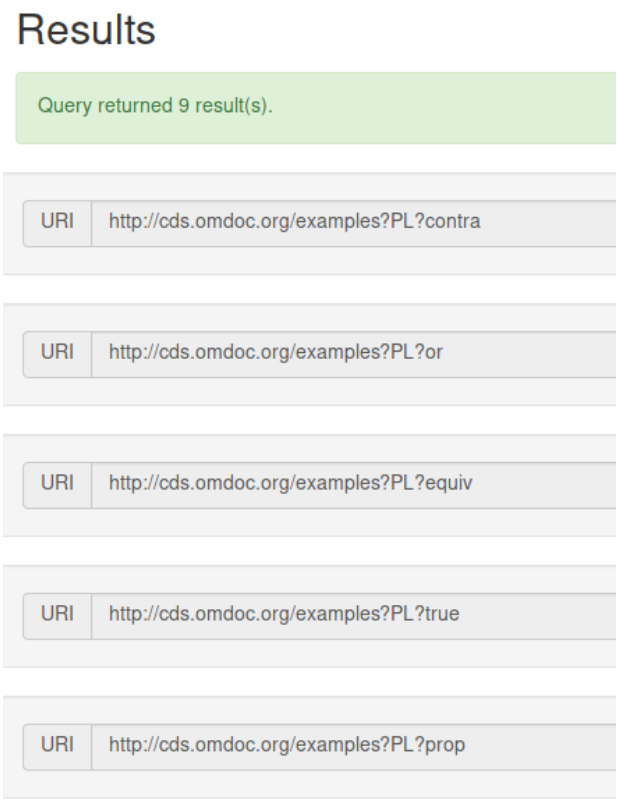


Figure 28: QMT Web Interface for displaying results.

The display of results can be found in Figure 28. On the web interface this is located below the entering interface and only shows up once results have been received. The interface is very plain and only shows two pieces of information, the number of results and each individual result.

7 Conclusion

In this thesis, we have addressed the research question on how to connect different mathematical knowledge systems in a generic, efficient and scalable manner to enable transparent, semantics-aware, distributed computation.

To do this, we decided to make use of the Math-In-The-Middle paradigm. Using Virtual Theories, QMT and SCSCP we have created a solution that indeed provides a solution to this question. To achieve this we have had to address all three initial aspects.

A generic mechanism to manage and sync knowledge stored in MKS
Using LMFDB as an example, we have seen how to build a virtual theory layer acting as such a mechanism. Our core contribution here is the choice of using Codecs to ensure separation of semantic and realized types. This choice has turned out exactly right – enabling our mechanism to easily integrate with different systems while at the same time not being ad-hoc. This makes it different and better than existing approaches.

A System-Independent Query Language
We have furthermore built on top of the existing QMT language of the MMT system. Instead of choosing to use a query language that relies on a specific data structure we can now see that we made the right choice here.

QMT was designed from the ground up to enable formulation of representation-independent queries. Even though it required a refactoring and some extensions to properly deal with distributed computations, we have seen that this choice fits in well with the mechanism of codecs and almost naturally extends the knowledge representational aspects of the MiTM approach to computational ones.

A Lower-Level Communication Layer
For this layer, we choose to use the SCSCP protocol. Along with the other two layers, and especially inside this one, we can see how it was a good choice to build on a system-independent framework. The representation of objects using OpenMath or OMDoc/MMT terms enabled systems to easily expose their computational functionality. This reaffirms again that communication between different specialist systems is possible without having to rely on an industry standard of having to rely on ad-hoc solutions – a generic pattern like the one here is possible.

To reflect on our contribution further, we take a final look at Jane's initial example of cross-system communication.

7.1 Revisiting the Use-Case For Cross-System Communication

With all the pieces in place, we can now see how our architectural choices help Jane express her computational problem in QMT.

Example 29. Jane's example expressed as a QMT Query.

```
use "gap" for (
  use "lmfdb" for {*
    x in (
      related to ( literal ‘lmfdb:db/transitivegroups?group ) by (object declares)
    ) | holds x (
      x commutative x *=* true
    )
  *}
) map x => cyclic x)
```

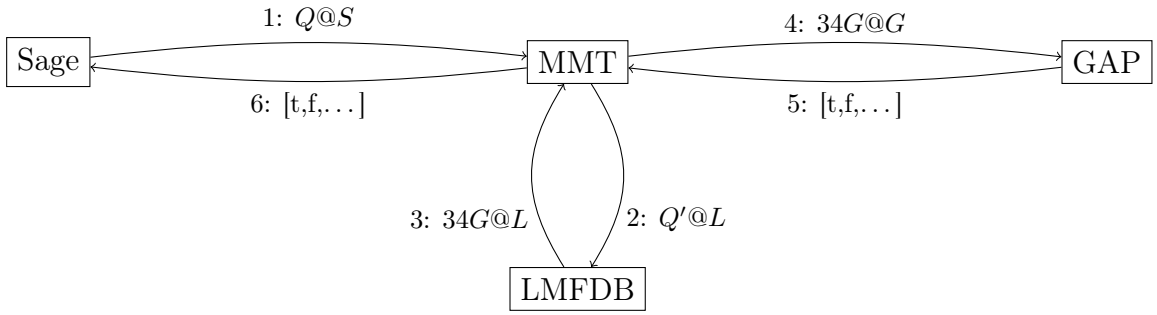


Figure 30: Elaborated technical details for Cross-System-Communication Use-Case. Compare with Figure 3.

We can now also have another look at Figure 3. In Figure 30 the steps of this process are annotated with more technical details.

1. Jane formulates a QMT Query which expresses this hypothesis and sends it to MMT. This is the one shown above and consists of two parts. The first part uses LMFDB to retrieve all abelian (commutative) groups from LMFDB. This sub-query corresponds to Example 22 and is in lines 2-8. The second part uses GAP to check if a group is indeed cyclic. Here, another `I()` query is used (line 1) to check if each of the results is indeed cyclic by using `map`.

2. MMT identifies the inner LMFDB query, translates and sends it to the LMFDBAPI.

3. MMT receives a list of 34 results from LMFDB and translates these back into Math-In-The-Middle terms.

4. MMT then translates these intermediate results into objects that GAP can understand. Next it translates the remainder of the query, and sends it to GAP along with the intermediate results using the SCSCP protocol.

5. GAP responds to the query and sends back a list of 34 boolean (OpenMath) objects. This lists contains both `true` and `false`, because it turns out that the original hypothesis is false.

6. Finally, MMT translates this set of results into a Sage objects which are then delivered back to Jane.

Most of these steps are hidden from Jane – she only sends a single QMT Query. This makes this example truly a transparent, distributed cross-system communication.

As we have seen the approach maintains object semantics across systems – this is illustrated by Jane never seeing a 0 or a 1 which is stored inside of LMFDB. Furthermore, she never knows that the abelian is stored in LMFDB as the `ab` property – no insight into LMFDB is required.

7.2 Outlook

Currently, steps 1 - 5 in are implemented inside of MMT and are fully functional. This once again shows that the MiTM approach is suitable of facilitating cross-system communication and was a valid choice. Furthermore, the approaches above are indeed feasible and improve on existing systems.

However, work in this area is far from complete. Apart from extending the approach to include more systems and testing it in more settings, there are a several improvements that can be made.

Consider for example QMT. Currently, sub-queries need manual annotation to mark them as being used with external systems. During this thesis it has been speculated that an automatic approach intercepting each intermediate result would be to computationally expensive. In a future extension of the query language, one might instead consider analyzing the structure of each query and automatically annotating parts of the query to be evaluated with different systems.

Furthermore, as stated above, step 6 of the example has not yet been implemented. This means making QMT available to the Sage system. Thus we need to implement one of two approaches, either make a Sage specific API, or exposing the QMT interface via SCSCP and building a client inside of Sage. The latter of these options seems appealing, and has been hinted on during this thesis. However, this approach requires more thought than might initially seem necessary.

It is straightforward to send queries from Sage, however it is not as easy to receive the results. As stated above, these need to be translated into something that Sage specific. Hence, when posing the Query, MMT needs to be made aware that the Query is coming from Sage. To implement this in a generic fashion that can eventually be extended to systems beyond the ones mentioned here, one could either add a separate case to the query language, or build this into an argument to be passed only via SCSCP.

7.3 Acknowledgements

The author acknowledges that the work for this project has been financially supported by the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541).

Furthermore, the author expresses his thanks to John Cremona for his help on understanding the structure of objects inside of LMFDB as well as all members of the KWARC group, and in particular Michael Kohlhase, for their input and help during the design and implementation phases of the work described here.

References

[Bus+04] *The OpenMath Standard, Version 2.0*. Tech. rep. The OpenMath Society, http://www.openmath.org/. 2004.

[D3.317] Steve Linton Luca De Feo Alexander Konovalov and Tom Wiesing. *Support for the SCSCP interface protocol in all relevant components (SAGE, GAP etc.) distribution*. Deliverable D3.3. OpenDreamKit, 2017. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/WP3/D3.3/report-final.pdf.

[Deh+16] Paul-Olivier Dehaye, Mihnea Iancu, Michael Kohlhase, Alexander Konovalov, Samuel Lelièvre, Dennis Müller, Markus Pfeiffer, Florian Rabe, Nicolas M. Thiéry, and Tom Wiesing. "Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics. (Bialystok, Poland, July 25–29, 2016). Ed. by Michael Kohlhase, Moa Johansson, Bruce Miller, Leonardo de Moura, and Frank Tompa. LNAI 9791. Springer, 2016. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf.

[Dev] The Sage Developers. *SageMath, the Sage Mathematics Software System*. http://www.sagemath.org. [Online; accessed 30 August 2016]. URL: http://www.sagemath.org.

[Fre+09] Sebastian Freundt, Peter Horn, Alexander Konovalov, Steve Linton, and Dan Roozemond. *Symbolic Computation Software Composability Protocol (SCSCP)*. Version 1.3. Mar. 27, 2009. URL: http://www.symcomp.org/SCSCP.

[GAP16] The GAP Group. *GAP – Groups, Algorithms, and Programming*. http://www.gap-system.org. [Online; accessed 30 August 2016]. 2016. URL: http://www.gap-system.org.

[How80] William A. Howard. "The formulae-as-types notion of construction". In: *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Hitherto unpublished note of 1969, rearranged, corrected and annotated by Howard, 1979. Academic Press, 1980, pp. 479–490.

[Ian17] Mihnea Iancu. "Towards Flexiformal Mathematics". PhD thesis. Bremen, Germany: Jacobs University, 2017.

[Inc] OEIS Foundation Inc., ed. *The On-Line Encyclopedia of Integer Sequences*. URL: http://oeis.org (visited on 05/28/2013).

[JSON] *JSON (JavaScript Object Notation)*. seen March 2009. URL: http://json.org/.

[KMR] Michael Kohlhase, Till Mossakowski, and Florian Rabe. *LATIN: Logic Atlas and Integrator*. URL: http://latin.omdoc.org (visited on 09/15/2010).

[KR14] Cezary Kaliszyk and Florian Rabe. "Towards Knowledge Management for HOL Light". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). Ed. by Stephan Watt, James Davenport, Alan Sexton, Petr Sojka, and Josef Urban. LNCS 8543. Springer, 2014, pp. 357–372. ISBN: 978-3-319-08433-6. URL: http://kwarc.info/frabe/Research/KR_hollight_14.pdf.

[LMF] The LMFDB Collaboration. *The L-functions and Modular Forms Database*. http://www.lmfdb.org. [Online; accessed 27 August 2016].

[MH] *MathHub.info: Active Mathematics*. URL: http://mathhub.info (visited on 01/28/2014).

[Miz] *Mizar*. URL: http://www.mizar.org (visited on 02/27/2013).

[MMTa] *MMT – Language and System for the Uniform Representation of Knowledge*. project web site. URL: https://uniformal.github.io/ (visited on 08/30/2016).

[MMTb] Florian Rabe. *The MMT System*. URL: https://uniformal.github.io/doc/ (visited on 07/16/2014).

[ODKa] *Open Digital Research Environment Toolkit for the Advancement of Mathematics*. Project Proposal. URL: https://github.com/OpenDreamKit/OpenDreamKit/raw/master/Proposal/proposal-www.pdf (visited on 09/01/2015).

[ODKb] *OpenDreamKit Open Digital Research Environment Toolkit for the Advancement of Mathematics*. URL: http://opendreamkit.org (visited on 05/21/2015).

[Pfe91] Frank Pfenning. "Logic Programming in the LF Logical Framework". In: *Logical Frameworks*. Ed. by Gérard P. Huet and Gordon D. Plotkin. Cambridge University Press, 1991.

[PVS] *NASA PVS Library*. URL: http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/ (visited on 12/17/2014).

[Rab09] F. Rabe. "The MMT Language". 2009.

[Rab11] Florian Rabe. *The MMT Language and System*. Oct. 11, 2011. URL: https://svn.kwarc.info/repos/MMT/doc/html (visited on 10/11/2011).

[Rab12] Florian Rabe. "A Query Language for Formal Mathematical Libraries". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 142–157. ISBN: 978-3-642-31373-8. arXiv: 1204.4685 [cs.LO].

[SMG] *SMGloM: A Semantic, Multilingual Terminology for Mathematics*. URL: http://smglom.mathhub.info (visited on 04/21/2014).

[Wol17] Wolfram Research Inc. *Wolfram Mathematica 11*. 2017. URL: https://www.wolfram.com/mathematica/.