REGULAR-T1: Virtual Theories – A Uniform Interface to Mathematical Knowledge Bases

Tom Wiesing[1] Michael Kohlhase[1] Florian Rabe[2]

[1] FAU Erlangen-Nürnberg
[2] Jacobs University Bremen

Abstract. If we want to support mathematical research, engineering, and education by computer systems, we need to deal with the various kinds of mathematical content collections and information systems available today. Unfortunately, these systems– ranging from Wikipedia to theorem prover libraries are usually only accessible via a dedicated web information system or a low-level API at the level of the raw database content. What we would want is a "programmatic, mathematical API" which would give access to the knowledge-bases programmatically via their mathematical constructions and properties.
This paper takes a step into this direction by interpreting knowledge bases as OMDoc/MMT theory graphs – modular, flexiformal representations of mathematical objects, their properties, and relations. For this we update OMDoc/MMT theories to "virtual theories" and update knowledge management algorithms so that they can cope with theories that do not fit into main memory but directly deal with the underlying databases as backends employing a modular system of codecs to bridge the gap between the database schema and the mathematical construction of objects.

1 Introduction

There are various large-scale sources of mathematical knowledge. These include
- generic information systems like the Wikipedia,
- collections of informal but rigorous mathematical documents – e.g. research libraries, publisher's "digital libraries", or the Cornell preprint arXiv,
- literature information systems like zbMATH or MathSciNet,
- databases of mathematical objects – like the GAP group libraries, the Online Encyclopedia of Integer sequences (OEIS), and the L-functions and Modular Forms Database (LMFDB),
- fully formal theorem prover libraries like those of Mizar, Coq, PVS, and the HOL systems. We commonly refer to all these as Mathematical Knowledge Bases.

We will use the term **mathematical knowledge bases** to refer to them collectively and restrict ourselves to those that are available digitally. They are very useful in mathematical research, applications, and education. Commonly these systems are only accessible via a dedicated web interface that allows humans to query or browse the databases. A programmatic interface, if it exists at

all, is system specific, meaning that to use it mathematicians need to be familiar both with the mathematical background and internal structure of the system in question. No predominant standard exists, and these interfaces usually only expose the low-level raw database content.

In this paper, we focus on addressing this problem. We ask the question of what mathematicians desire from a "programmatic, mathematical API". Such an API would give access to the knowledge-bases programmatically via their mathematical constructions and properties.

In this paper, we take it a step further and discuss our implementation of such an approach. We interpret mathematical knowledge bases as OMDoc/MMT theory graphs – modular, flexi-formal representations of mathematical objects, their properties, and relations. This embedding gives us a common conceptual framework to handle different knowledge sources, and the modular and heterogeneous nature of OMDoc/MMT theory graph can be used to reconcile differing ontological commitments of the knowledge sources with in this conceptual framework.

To cope with the scale of mathematical content collections we update OMDoc/MMT theories to "virtual theories", which no longer limit the number of declarations in theory and practice, and update knowledge management algorithms in the MMT system so that they can cope with theories that do not fit into main memory but directly deal with the underlying databases as backends employing a modular system of codecs to bridge the gap between the database schema and the mathematical construction of objects.

This paper proceeds as follows: In Section 2 we give a short overview of OMDoc/MMT theory graphs along with the Math-In-The-Middle approach developed in the OpenDreamKit project, our primary use-case for Virtual Theories. We then continue in Section 3 by giving an example a State-Of-The-Art Mathematical Database along with its' interface by discussing the LMFDB. In Section 4 we then describe how to represent this example as a set of Virtual Theories. We move on in Section 5 to describe how to access Virtual Theories using our codec architecture. Section 6 concludes the paper.

2 Virtual Research Environments for Mathematics: the Math-in-the-Middle Approach

The work reported in this paper originates from in the EU-funded OpenDreamKit [ODK] project that aims to create Virtual Research Environments (VRE) enabling mathematicians to make efficient use of existing Open-Source mathematical knowledge systems. These systems include computer algebra systems like SageMath and GAP as well as mathematical data bases such as the LMFDB, which must be made interoperable for integration into a VRE. In the OpenDreamKit project we have developed the Math-in-the-Middle (MitM) approach, which posits a central ontology of mathematical knowledge, which acts as a pivot point for interoperability; see [Deh+16] for a description of the approach and [Koh+] for a technical refinement and large-scale interoperability case study.

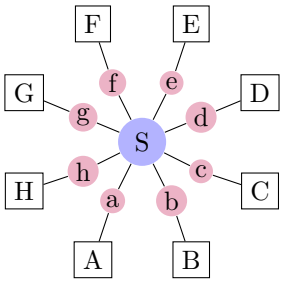

Fig. 1. The MiTM Approach to Connecting Systems.

The MitM ontology in the center of Figure 1 models the true, underlying mathematical semantics in OMDoc/MMT and allows translation between this centrally formalized knowledge and the systems on the boundary via views and alignments. This mathematical knowledge is modeled using the well-established theory graph paradigm and is stored inside our OMDoc/MMT-based MathHub system [MH].

The knowledge in the mathematical software systems – denoted by square boxes in Figure 1 – also modeled via OMDoc/MMT theory graphs the **API Content Dictionaries** – the corresponding red circles; these are generated from the knowledge bases in the systems by a custom process. The API CDs allow us to implement translation with the help of OMDoc/MMT views and alignments between the ontology – the Math-In-The-Middle – and each of the systems and use these translations for transporting computational tasks between the systems.

The realization of the MitM approach crucially depends on the information architecture of the OMDoc/MMT language [Koh06; RK13] and its implementation in the MMT system [Rab13; MMT].

In OMDoc/MMT knowledge is organized in **theories**, which contain information about mathematical concepts and objects in the form of **declarations**. Theories are organized into an "object-oriented" inheritance structure via **inclusions** and **structures** (for controlled multiple inheritance), which is augmented via truth-preserving mappings between theories called **views**, which allow to relate concepts of pre-existing theories and transport theorems between these. Inclusions, structures, and views impose a graph structure on the represented mathematical knowledge, called a **theory graph**.

We observe that even very large mathematical knowledge spaces about abstract mathematical domains can be represented by small, but densely connected, theory graphs, if we make all inherited material explicit in a process called **flattening**. The OMDoc/MMT language provides systematic names (MMT URIs) for all objects, properties, and relations in the induced knowledge space, and given the represented theory graph, the MMT system can compute them by need.

Generally, knowledge in a knowledge space given by a theory graph loaded by the MMT system can be accessed by either giving it's MMT URI, or by giving a set of conditions that has to fulfilled by the knowledge in question. The achieve the latter, MMT has a Query Language called QMT [Rab12], which allows even complex conditions to be specified. Concretely, the MMT system loads the theory graph into main memory at startup and interleaves incremental flattening and query evaluation operations on the MMT data structures until the result has been produced.

In [Koh+] we show that the MitM approach, its OMDoc/MMT-based realization, and distribution via the SCSCP protocol are sufficient distributed,

federated computation between multiple computer algebra systems (Sage, GAP, and Singular), and that the MitM ontology of abstract group theory can be represented in OMDoc/MMT efficiently. This setup is effective because

C1 the knowledge spaces behind abstract and computational mathematics can be represented in theory graphs very space-efficiently: The compression factors between a knowledge space and its theory graph – we call it the **TG factor**[1] – exceed two orders of magnitude even for small domains.

EdN:1

C2 only small parts of the knowledge space are traversed for a given computation.

But the OpenDreamKit VRE must also include mathematical data sources like the LMFDB or the OEIS, which contain millions of mathematical objects. For such knowledge sources, the classical MMT system is not yet suitable:

V1 the knowledge space corresponding to the data base content cannot be compressed by "general mathematical principles" like inheritance. Indeed, redundant information is already largely eliminated by the data base schema and the "business logic" of the information system.

V2 typically large parts of the knowledge space need to be traversed to obtain the intended results to queries.

Therefore, we extend the concept of OMDoc/MMT theories – which carry the implicit assumption of containing only a small number of declarations (see [FGT92] for a discussion) – to **virtual theories**, which can have an unlimited (possibly infinite) number of declarations. To contrast the intended uses we will call the classical OMDoc/MMT theories **concrete theories**. In practice, a virtual theory is represented by concrete approximations: OMDoc/MMT works with a concrete theory, whose size changes dynamically as a suitable backend infrastructure generates declarations on demand.

3 Example: The API and Structure of LMFDB

The "L-functions and Modular Forms Database" (LMFDB [LMF]) is a Python web application with a MongoDB backend. The project contains several thousand L-Functions and curves along with their properties. We use this as an example of a Virtual Theory. Before we go into this in more detail, we first have a closer look at the structure and existing APIs to communicate with it.

3.1 The Structure of LMFDB

LMFDB has several sub-databases – each of which contains different kinds of objects.

These databases include e.g. a database of elliptic curves or a database of transitive groups. Within each database, each curve is stored as a single JSON

[1] EdNote: MK@FR: we should have a name for that factor; something like the "deBruijn factor"; only that we want it to be large! I think we should talk about this on the tetrapod meeting and jointly decide on one and argue with that; I will use TG factor for now.

record with common keys, Figure 2 shows one: each property of this JSON object corresponds to a property of the underlying mathematical object. For example, the `degree` property – here 1 – of the JSON objects corresponds to the degree of the underlying elliptic curve.


```
{
    "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')"
}
```


Fig. 2. An elliptic curve, as found within LMFDB. Some key-value pairs are omitted for readability.

Other properties are more complex. Whereas the value of the `degree` property is a simple integer, the value of the `isogeny_matrix` property is a list of lists, which represents a matrix. This can become even more technical. For example the `x-coordinates_of_integral_points` field, LMFDB represents a list of integers as a list of strings as the integers can exceed the range of MongoDB system integers. This already shows that it is non-trivial to get from a MongoDB encoding of an elliptic curve in LMFDB to the representation of a mathematical object.

3.2 An API for LMFDB Objects

As LMFDB is is a mathematical knowledge base, one important use case is to find elliptic curves subject to specific criteria. Consider for example a mathematician that wants to find all abelian elliptic curves in LMFDB. How can this be achieved using the LMFDB API located at [Lmf]?

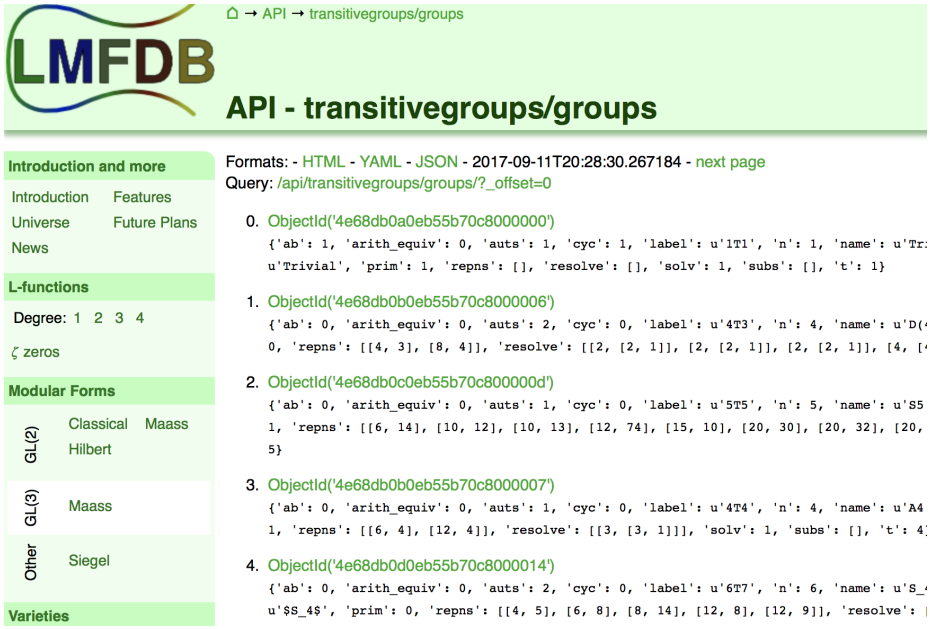

Fig. 3. The Web-Interface for the LMFDB API.

Queries can be sent to the API by making appropriate GET requests. The LMFDB API can present results in two different ways, either using a web-based interface or programmatically by returning a set of JSON objects. A screenshot of the former can be seen in Figure 3. The mode can be decided upon by adding an appropriate parameter to the query. In the following we will focus on the latter mode only, however all links will not include this format parameter so that readers can follow along in the web browser[2].

EdN:2

Queries must be formulated in terms of the underlying MongoDB schema, are sub-database specific, and should consist of a set of key value pairs. To solve the example given here we need to send the key-value pair `commutative=true`, finding all elements for which the commutative property is true. However, these values need to be encoded to be understood by MongoDB. We need to realize that the `ab` key corresponds to the commutativity property, has boolean

[2] EdNote: Since the API is broken, I'm not sure what to do here

values, and that MongoDB encodes `true` as `1`, and `false` as `0` in this LMFDB sub-database. This information can then be used to make a query by sending a request to `http://www.lmfdb.org/api/transitivegroups/groups/?ab=1`.

In this example, each of the steps are relatively straightforward. In a general setting, e.g. when searching for all elliptic curves with a specific isogeny matrix, this not only requires a good familiarity with the mathematical background but also with the system internals of the particular LMFDB sub-database; a skillset commonly found in neither research programmers nor average mathematicians.

To summarize: while LMFDB offers a programmable API for accessing its contents, the content API is at the MongoDB level, and not the level of mathematical objects. Our Diagnosis is that LMFDB– and most other mathematical knowledge databases – suffer from a double impedance mismatch problem.

I1 *human/computer impedance mismatch*: Humans have problems interacting with LMFDB, since they must speak the system language instead LMFDB speaking mathematics

I2 *computer/computer impedance mismatch*: mathematical computer systems cannot interoperate, since their system languages differ.

In the MitM approach we have presented in Section 2, we can solve both problems at the same time by lifting the communication to the level of OMDoc/MMT-encoded MitM objects, which both MitM-compatible software systems and humans speak – this is the central assumption of the MitM approach.

4 LMFDB as a Set of Virtual Theories

The mathematical software systems to be integrated via the MitM approach have so far been computation-oriented, e.g. computer algebra systems. Their API CDs typically declare types and functions on these types (the latter including constants seen as nullary functions). Even though database systems differ drastically from these in many respects, they are very similar at the MitM level: a database like LMFDB defines

- some types: each table's schema is one type definition,
- many constants: each entry of each table is one constant of the corresponding type.

Thus, we can apply essentially the same approach. In particular, the API CDs must contain definitions of the database schemas.

From a system perspective, virtual theories behave just like concrete theories, but without the assumption of loading all declarations from a file on disk at system startup. Instead, virtual theories load declarations in a lazy fashion when they are needed. Concrete theories are stored as XML files; i.e. we use the file system as a backend for the MMT system. As most of the knowledge sources we want to embed into OMDoc/MMT as virtual theories use data-bases as backends and provide low-level database APIs we have extended the MMT backends for this as well. Apart from standard software engineering tasks, there were three conceptual problems to be solved in this extension/implementation:

P1 How to match the database tables into OMDoc/MMT theories and declarations.

P2 How to lift data in **physical representation** – i.e. as records of the underlying database to OMDoc/MMT terms – i.e. data in **semantic representation**.

P3 And how to translate QMT queries from semantic to physical representation – i.e. so that they can be executed directly on the data base without loading bulk data into the MMT process.

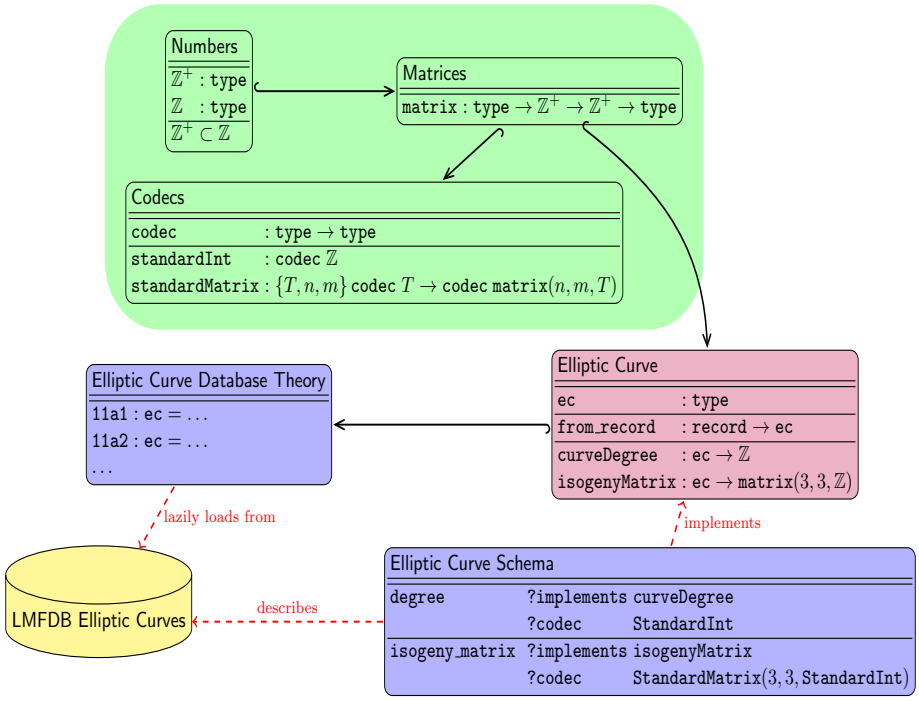

Fig. 4. A sketch of the architecture for a virtual theory connecting to LMFDB. Solid edges represent imports. Several declarations have been omitted for simplicity.

A sketch of our overall soluton can be seen in Figure 4. We start by dealing with **P1**. The set of constants in a database table – while finite – can be arbitrarily large. In particular, all LMFDB tables[1]are just finite subsets of infinite sets, whose size is not limited by mathematical specifications but by computational power: the database holds all objects that users have computed so far and grows constantly as more objects are computed. LMFDB tables usually include a naming system that defines unique identifiers (which are used the database keys) for these objects, and these identifiers are predetermined even for those

objects that have not been computed yet. Thus, it is not desirable to fix the a set of concrete API CDs. Instead, the API CDs must split into two parts: for each database table, we need

- a concrete theory called the **schema theory** that defines the schema and other relevant information about the type of objects in the table and
- a virtual theory called the **database theory** that contains one definition for each value of that type (using the LMFDB identifier as the name of the defined constant).

Both of these can be found colored in blue in Figure 4, the schema theory on the bottom right, the database theory in the middle on the left.

We address **P2** next. LMFDB's technical realization does not require formalizing the schema of each table. Instead, the tables are generated systematically and therefore follow an implicit schema that can – in principle – be obtained from the documentation or reverse-engineered from the tables. However (and here LMFDB critically differs from, e.g. OEIS), the mathematics involved in the tables so deep that this is not possible in practice for all but a few experts. Therefore, we sat down with the original author of one of the best-documented tables – John Cremona for the table of elliptic curves – and formalized the corresponding schema in OMDoc/MMT.

In the following, we will use this table as a running example. Our methods extend immediately to any other table once its schema has been formalized.

Before we can continue however, we first need to have a closer look at our formalisation of *Elliptic Curve*s which can be seen in the red in Figure 4.

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[2] of elliptic curves.

These are *degree*, an integer, and the *isogeny matrix*, a 3×3 matrix of integers. They 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, so that we can integrate other knowledge sources about elliptic curves or to future versions of the LMFDB with changed structure.

[1] Technically, LMFDB is implemented using MongoDB and comprises a set of sets (each one called a database) of JSON objects. However, due to the conventions used, we can also understand it conceptually as a set of tables of a relational database, keeping in mind that every row is a tuple of arbitrary JSON objects.

[2] 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.

5 Accessing Virtual Theories

Intuitively, it is straightforward how to fill a virtual theory V: V is represented by an initially empty concrete theory V', and whenever an identifier id of V is requested, MMT dynamically adds the corresponding declaration of id to V'. MMT already abstracts from the physical realizations of persistent storage using the *backend* interface: essentially a backend is any component that allows loading declarations. Thus, we only have to implement a new backend that connects to LMFDB, retrieves the JSON object with identifier id, and turns into an OMDoc/MMT declaration.

However, this glosses over a major problem: the databases used for the physical storage of large datasets usually relatively simple data structure. For example, a JSON database (as underlies LMFDB) offers only limited-precision integers, boolean, strings, lists, and records as primitive objects and does not provide a type system. Consequently, the objects stored in the database are very different from the sophisticated mathematical objects expected by the schema theory. Therefore, databases like LMFDB must encode this complex mathematical objects as simple database objects.

5.1 Concrete Encodings of Mathematical Objects

Consider, for example the `degree` field from Figure 2 above. It's value is the integer 1, representing the degree of this curve. However inside the database it is represented as the `IEEE754` 64-bit floating point number 1.0. But when the semantic representations can exceed the have a maximum possible value $2^{53} - 1$ of IEEE floats, LMFDB needs to use a different encoding, e.g. JSON strings that have no hard upper limit.

Let us call the set of objects in semantic representation the **semantic type**, and the set of objects in the physical representation the **realized type**. Semantic types reside in the MiTM ontology, whereas realized types reside in the systems themselves. Corresponding with intuition, 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**. We will call system components that do the necessary translation – coding and decoding – **CoDec**s.

As OMDoc/MMT is a typed framework, we can directly use OMDoc/MMT types from the MitM ontology for the semantic types. Simple realized types are usually atomic database types whereas complex realized types correspond to database tables or views; the details of this setup are determined by the database schema. To arrive at a tight integration with the OMDoc/MMT functionality we will represent as much of this information in OMDoc/MMT as possible.

Therefore we introduce a new OMDoc/MMT theory `Codecs` in the foundational part of the MitM ontology. See the green part of Figure 4 for details how this plays in the overall information architecture and Figure 5 for elementary content. This theory introduces a type constructor `codec`, which given a semantic type constructs the type of CoDecs for this type. For instance, the object `StandardPos` is (a CoDec) of type `codec` $\mathbb{Z}^+$, i.e. a CoDec that parses database

objects (for LMFDB IEEE floats) into OMDoc/MMT terms that can be typed as MitM positive integers and serializes them back.

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

Fig. 5. An annotated subset of the 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 `degree` we used as an example above would use the `StandardInt` CoDec. Additionally the CoDecs theory associates with each codec a Scala class that implements the translation between semantic and realized type.

But CoDecs for basic types (semantic and realized) are not sufficient for our application. Consider for example the `isogeny_matrix` field of an elliptic curve representation. The semantic representation of the value of this field is the matrix

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

Matrices are characterized with three 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.

Instead we use the fact that both Scala and OMDoc/MMT allow higher-order functions: We can define a **codec operator** that given a CoDec the parameter type τ and values for the number n of rows and m of columns, generate a CoDec of $n \times m$ matrices of τ objects. In the example above and the matrix M is encoded as a list of n lists of m integers (τ):

```
[[1.0,5.0,25.0],[5.0,1.0,5.0],[25.0,5.0,1.0]]
```

Like first-order CoDecs, CoDec operators in MMT are again represented in two ways, as declarations inside the `CoDecs` theory (see Figure 6 for a list, also compare again with Figure 4) and as a corresponding Scala implementation –

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

Fig. 6. Second annotated subset of the CoDecs theory containing a selection of CoDec operators found in MMT. Compare with Figure 5.

a higher-order function from CoDecs to CoDecs. This is mirrored in the types of operators in Figure 5, the StandardMatrix operator is a function that takes four arguments: a type T, two numbers n and m, and a τ-CoDec and yields a Matrix(n, m, T)-CoDec. Here we make use of the dependent function types of the MitM foundation: arguments in curly brackets can be used in the result type; see [RK13] for details.

With these declarations in the CoDecs theory, we can represent a CoDec for 3×3 integer matrices e.g. for the the isogeny matrix M above by the OMDoc/MMT term `StandardMatrix`$(3, 3,$ `StandardInt`$)$. Similarly the same CoDec operator can be used to for example generate a CoDec for 2×2 boolean matrices, which corresponds to `StandardMatrix`$(2, 2,$ `StandardBool`$)$.

5.2 Specifying Encodings in Schema Theories

Given this infrastructure, let us see how we can integrate knowledge sources like the LMFDB in the Math-In-The-Middle approach. 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 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.

The database theory is the truly virtual theory – it is not stored on disk, but generated dynamically. As designed, it contains one declaration per record in LMFDB. It uses an MMT **backend** – an MMT abstraction used to load declarations into memory. Given a URI, the backend is responsible for loading the underlying definition. For the elliptic curve theories these URIs are of the form `lmfdb:db/elliptic_curves?curves?11A1`.

The backend first retrieves the appropriate record from LMFDB – in the case of `11A1` this corresponds to retrieving the JSON found in Figure 2. 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. Notice how even though the physical and semantic representations are shown identically here, they are indeed different: The former is a 64-bit floating point JSON Number, the latter is a mathematical integer.

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 $\texttt{StandardMatrix}(3, 3, \texttt{StandardInt})$. 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 of M.

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.

5.3 Translating Queries

Recall that MMT has a Query Language called QMT [Rab12], which allows users to find knowledge to complex conditions to be specified. We continue by briefly addressing **P3**, however for a complete discussion we refer the interested reader to [Wie17].

In practice, most queries involving virtual theories so far have a shape similar to the one that LMFDB supports: Finding all objects within a single sub-database for which a specific field equals a specific value. As an example, consider again

the query of finding all abelian transitive groups. This can be expressed in QMT as: [3]

EdN:3

Recall that to evaluate a query prior to the introduction of Virtual Theories, the MMT system loaded the theory graph into main memory and then interleaved incremental flattening and query evaluation operations on the MMT data structures until a result has been produced. This can no longer be applied to resolve the query above, as not all relevant data is present in memory. Moreover, it is also not feasible to first load all potentially relevant data into memory, and only then proceed with evaluation. This would require loading a copy of LMFDB into main memory, something that virtual theories were designed to avoid.

As we have already seen, LMFDB has an API. This API is in principle capable of efficiently resolving the query shown here, however it does not directly support the QMT Query Syntax but requires translation first. In general, most mathematical knowledge bases have a similar API serving as an information retrieval mechanism – commonly in the form of a query language or an API.

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. As Queries are inductive in nature, one could attempt to intercept each of the intermediate results. However, this would require a check on each intermediate result to first determine if it comes from a virtual theory or not, and then potentially switching the entire evaluation strategy, leading to a very computationally expensive implementation.

Instead of intercepting each result, we extended the Query Language to allows users to annotate sub-queries for evaluation with a specific API. This allows the system to immediatly know which parts of a query have to be evaluated in MMT memory, and which have to be translated and sent to an external API. This new approach turn the example above into: [4]

EdN:4

[3] EdNote: TW: Actually add the example of the query (without I()); do a very brief description of it

[4] EdNote: TW: Add the example again, this time with I(), then do a bit more explaining

6 Conclusion

We have shown how to extend the data model for theories of the MMT system, and implemented a generic approach that does no longer require the theories to reside in the main memory; instead knowledge is retrieved from an external system and declarations are generated on demand. The main conceptual leap here was to make a difference between the representations of objects in the database, and the underlying mathematical objects, and to translate between them using generic codecs. We have demonstrated that this approach is functional using the example of LMFDB.

The main advantages of this approach are, unlike existing interfaces to knowledges bases, its' generic design and easy extensibility.

- To add a new LMFDB database to the set of represented theories one in practice only has to add a new schema theory. This schema theory will have to contain the sets of fields within this new database, along with their mathematical types and codecs. All of these should be known to the database maintainers, albeit not directly as types and `codecs`, and are thus easy to find. This schema theory can then be used to automatically generate a new database theory.
- To add a knowledge base that uses a different data base, we also have to implement a new MMT backend and – possibly – extend the set of CoDecs.

There are several other aspects which we have not detailed here as addition of Virtual Theories to MMT impacts several other aspects of the system. For example, MMT has a Query Language allowing users to query information available; the previous implementation relied on theories being concrete, which is no longer true for all theories. Furthermore, we are also planning to extend our implementation of Virtual Theories; e.g. we want to extend it to more than a few LMFDB databases. We also have concrete plans for a second example based on OEIS[5]. EdN:5

Acknowledgements The authors gratefully acknowledge the fruitful discussions with other participants of work package WP6, in particular John Cremona on the LMFDB and Jörg Arndt on the OEIS. We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541).

References

[Deh+16] Paul-Olivier Dehaye et al. "Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach". In: *Intelligent Computer Mathematics 2016*. Ed. by Michael Kohlhase et al. LNAI 9791. Springer, 2016. URL: `https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf`.

[5] EdNote: We had at some point at least

[FGT92] William M. Farmer, Josuah Guttman, and Xavier Thayer. "Little Theories". In: *Proceedings of the 11th Conference on Automated Deduction*. Ed. by D. Kapur. LNCS 607. Saratoga Springs, NY, USA: Springer Verlag, 1992, pp. 467–581.

[Koh+] Michael Kohlhase et al. "REGULAR-T1: Knowledge-Based Interoperability for Mathematical Software Systems". submitted to MACIS-2017. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/MACIS17-interop/submit.pdf.

[Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Lmf] *LMFDB - API*. http://www.lmfdb.org/api/. visited on: 09/17/2017.

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

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

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

[Rab12] Florian Rabe. "A Query Language for Formal Mathematical Libraries". In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 142–157. arXiv: 1204.4685 [cs.LO].

[Rab13] Florian Rabe. "The MMT API: A Generic MKM System". In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. Lecture Notes in Computer Science 7961. Springer, 2013, pp. 339–343. DOI: 10.1007/978-3-642-39320-4.

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[Wie17] Tom Wiesing. "Enabling Cross-System Communication Using Virtual Theories and QMT". Master's Thesis. Bremen, Germany: Jacobs University Bremen, Aug. 2017. URL: https://github.com/tkw1536/MasterThesis/raw/master/thesis.pdf.

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