Report on OpenDreamKit deliverable D6.5 GAP/SAGE/LMFDB Interface Theories and Alignment in OMDoc/MMT for System Interoperability

John Cremona

Dennis Müller

Michael Kohlhase

David Lowry-Duda

Markus Pfeiffer

Florian Rabe

Nicolas M. Thiéry

Tom Wiesing

ABSTRACT. There is a large ecosystem of mathematical software systems and knowledge bases. Individually, these are optimized for particular domains and functionalities, and together they cover many needs of practical and theoretical mathematics. However, each system specializes on one particular area, and it remains very difficult to solve problems that need to involve multiple systems. Some integrations exist, but they are ad-hoc and have scalability and maintainability issues. In particular, there is not yet an interoperability layer that combines the various systems into a virtual research environment (VRE) for mathematics.

The OpenDreamKit project aims at building a toolkit for such VREs. It suggests using a central system-agnostic formalization of mathematics (Math-in-the-Middle, MitM) as a mathematical pivot point for semantic-preserving translations in the needed interoperability layer. In this report, we report on a series of case studies that instantiates the MitM paradigm with the systems GAP, SageMath, LMFDB, and Singular to perform distributed computation in group, ring, and number theory.

Our work involves massive practical efforts, including a novel formalization of computational group theory, improvements to the involved software systems, an extension of the underlying knowledge managment system to cope with large theories, and a novel mediating system that sits at the center of a star-shaped integration layout between mathematical software systems and knowledge bases.

Due on	1/09/2017
Delivered on	1/07/2018
Lead	Friedrich-Alexander Universität Erlangen/Nürnberg (FAU)

Progress on and finalization of this deliverable has been tracked publicly at: https://github.com/OpenDreamKit/OpenDreamKit/issues/139

Contents

1. Introduction

There is a large and vibrant ecosystem of open-source software systems for mathematics. These range from calculators, which perform simple computations, via mathematical databases, which curate collections of mathematical objects, to powerful modeling tools and computer algebra systems (CAS).

These systems can be very specific, often focusing on a narrow area of mathematics. For example, among databases, the "Online Encyclopedia of Integer Sequences" (OEIS) focuses on sequences over $\mathbb{Z}$ and their properties, and the "L-Functions and Modular Forms Database" (LMFDB) [Cre16; LMFDB] on objects in number theory pertaining to Langland's program. Among CAS, GAP [GAP] excels at discrete algebra with a focus on group theory, Singular [SNG] focuses on polynomial computations with special emphasis on commutative and non-commutative algebra, algebraic geometry, and singularity theory. Finally, SageMath [Sage] aims to be a general purpose software for computational pure mathematics by integrating many systems including the aforementioned ones, together with a large body of code for the SageMath library itself written in Python.

For a mathematician, however, (a user, which we call Jane) the systems themselves are not relevant. Instead, she only cares about being able to solve problems. Because it is typically not possible to solve a mathematical problem within a single system, Jane has to work with multiple systems and combine the results to reach a solution. Currently there is very little tool support for this in practice; so Jane has to isolate sub-problems that the respective systems are amenable to, formulate them in the respective input language, collect intermediate results, and reformulate them for the next system — a tedious and error-prone process at best, a significant impediment to scientific progress at worst. Solutions for some situations certainly exist, which can help get Jane unstuck, but these are ad-hoc and only for specific often-used system combinations. Moreover, each of these ad hoc solutions requires a lot of maintenance and scales badly to multi-system integration. To add insult to injury, the knowledge bases Jane would like to use — ranging from Wikipedia to theorem prover libraries — are usually only accessible via the restricted API of a dedicated web information system or the low-level API to 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.
[1]

Towards a Virtual Research Environment for Mathematics. One goal of the OpenDreamKit project is tackling these problems systematically by building virtual research environments (VRE) on top of the existing systems. To build a VRE from individual systems, we need a joint user interface — the OpenDreamKit project adopts Jupyter [Jup] and active documents [Koh+11] — and an interoperability layer that allows passing problems and results between the disparate systems. For the latter, it proposes the Math-in-the-Middle paradigm (MitM [Deh+16]): an interoperability framework based on a central, system-independent ontology of mathematical knowledge and system API theories that specify the interfaces of the various systems in the same, modular knowledge representation format. We use OMDoc/MMT as this format together with alignments the describe the relations between the functions declared in the ontology and those realized in the various system APIs.

Contribution. In this report we instantiate the MitM paradigm in two concrete case studies. In the first one, we show distributed computation involving the GAP, SageMath, and Singular

[1] EDNOTE: NT: one should say something about previous attempts, e.g. OpenMath, why it does not do the job, and how we mean to fix that.

systems. In the second one, we show the integration of the mathematical knowledge base LMFDB into MitM-based computation.[2] EdN:2

For the CAS case study, we will use the following running example from computational group theory: Jane wants to experiment with invariant theory of finite groups. She works in the polynomial ring $R = \mathbb{Z}[X_1, \ldots, X_n]$, and wants to construct an ideal I in this ring that is fixed by a group $G \leq S_n$ acting on the variables, linking properties of the group to properties of I and the quotient of R by I.

To construct an ideal that is invariant under the group action, it is natural to pick some polynomial p from R and consider the ideal I of R that is generated by all elements of the orbit $O = Orbit(G, R, p) \subseteq R$. For effective further computation with I, she needs a Göbner base of I.

Jane is a SageMath user and wants to receive the result in SageMath, but she wants to use GAP's orbit algorithm and Singular's Gröbner base algorithm, which she knows to be very efficient. For the sake of example, we will work with $n = 4$, $G = D_4$ (the dihedral group[1]), and $p = 3 \cdot X_1 + 2 \cdot X_2$, but our results apply to arbitrary values.

For the LMFDB case study, Jane wants to investigate the number fields which are generated by the coefficients of Hilbert modular forms (HMFs). For many totally real number fields F of low degree, and for many levels $\mathcal{N}$ for each field, the LMFDB contains information about all HMFs of level $\mathcal{N}$ (of parallel weight 2 and trivial character). Each of these HMFs is an eigenform for the Hecke algebra with eigenvalues generating a number field; the same number field contains the coefficients of the standard Fourier expansion of the HMF, which are expressible in terms of the eigenvalues. In the LMFDB, each HMF's Hecke field K is stored by means of a defining polynomial which has been obtained as a by-product of the computation of the HMF itself, and is in no way canonical or minimal, making study of these fields difficult and — even in simple cases — obscure. For example, the Hecke field $K = \mathbb{Q}(\sqrt{2})$ may occur for more than one HMF, defined by the polynomial $x^2 - 2$ for some and by the polynomial $x^2 - 2x - 1$ for others. Hence Jane would like to be able to extract these defining polynomials from the LMFDB, use them to define number fields in SageMath, find simpler polynomials defining the same fields, and study their arithmetic properties (for example, their class numbers). To this end, some of the Hecke fields may themselves be in the LMFDB's collection of number fields, in which case the information about them which Jane needs is already computed and stored, but this will in most cases be hidden since the defining polynomials used in the HMF database will often not be the one stored in the number fields database.

Overview. In Section 2, we recap the MitM paradigm. In Section 3, we describe the MitM ontology. While the ontology describes the *abstract* syntax of mathematical objects, Section 4 introduces a codec framework for describing their *concrete* syntax in different systems. In Section 5 and 6, we describe the integration of the computation systems GAP, SageMath, and Singular and the LMFDB databases with the MitM architecture. In Section 7, we present the resulting virtual research environment built on these systems in action. Section 8 concludes the paper.

2. The Math-in-the-Middle Approach

Figure 1 shows the basic MitM design. We want to make the systems A to H with system dialects a to h interoperable. A P2P translation regime ($n(n-1)$ translations between n systems) is already intractable for the systems in the OpenDreamKit project (more than a dozen).

[2]EDNOTE: maybe preview contributions of the MitM paradigm here

[1]Incidentally, this group is called D_4 in SageMath but D_8 in GAP due to differing conventions in different mathematical communities – a small example of the obstacles to system interoperability that MitM tackles.

Alternatively, an “industry standard” regime, where one system dialect is declared as the standard is infeasible because no system dialect subsumes all others – not to mention the political problems such a standardization would induce. Instead, MitM uses a central mathematical ontology that provides an independent mediating language, via which all participating systems are aligned. All mathematical knowledge shared between the systems and exposed to the high-level VRE user is expressed using the vocabulary of this ontology. Crucially, while every system dialect makes implementation-driven, system-specific design choices, the MitM ontology can remain close to the knowledge published in the mathematical literature, which already serves as an informal interoperability layer.

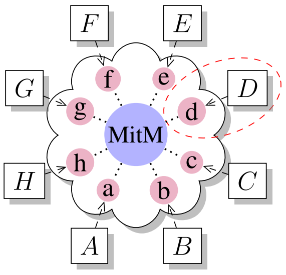


FIGURE 1. MitM Paradigm

The following sections describe the three components of the MitM paradigm in more detail.

2.1. The MitM Ontology

In the center, we have the **MitM Ontology**, which is a formalization of the mathematical knowledge behind the systems A to H. As a formalization framework, it uses the OMDoc/MMT format [Koh06; RK13; MMT], which was designed with this specific application in mind. We do not go into the details of OMDoc/MMT here – for our purposes, it suffices to assume that an OMDoc/MMT theory graph formalizes a language for mathematical objects as a set of typed symbols with a (formal or informal) specification of their semantics. For example, the MitM-symbol `PolynomialRing` takes a ring r of coefficients and a number n of variables and returns the ring $r[X_1, \dots, X_n]$ of polynomials.

Note that the purpose of the MitM ontology is not the formal verification of mathematical theorems (as for most existing formalizations such as [Gon+13] of group theory), but to act as a pivot point for integrating systems. This means that it can be much nearer to the informal but rigorous presentation of mathematical knowledge in the literature. While each system dialect makes compromises and optimizations needed for a particular application domain, the MitM ontology follows the existing and already informally standardized mathematical knowledge and can thus serve as a standard interface layer between systems.

Importantly, the MitM ontology does not have to include any definitions[2] or proofs – it only has to declare the types of all relevant symbols and state (but not prove) the relevant theorems. This makes it possible for users like Jane to extend the MitM ontology quickly whereas extending formalizations usually requires extensive efforts by specialists.

2.2. OMDoc/MMT as a Knowledge Representation Format

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

[2]Of course, definitions are one possible way to specify the semantics of MitM-symbols.

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 on demand.

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 uniquely describing it via a set of conditions. To achieve the latter, MMT has a Query Language called QMT [Rab12], which allows even complex conditions to be specified. Currently, 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 this report we show that the MitM approach, its OMDoc/MMT-based realization, and distribution via the SCSCP protocol are sufficient for 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

- 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** – exceeds two orders of magnitude even for small domains.
- only small parts of the knowledge space are traversed for a given computation.

2.3. Specifying System Dialects via System API Theories

System Dialects. It is unavoidable that each system induces its own language for mathematical objects. This is the cause of much incompatibility because even subtle differences make naive integration impossible. Moreover, due to the difficulty of the involved mathematics and the effort of maintaining the implementations, such differences are aplenty.

Fortunately, we can at least easily abstract from the user-facing surface syntax of these languages: scalable interoperability can anyway only be achieved by acting on the internal data structures of the systems. Thus, only the much simpler internal abstract syntax needs to be considered.

The symbols that build the abstract syntax trees can be split into two kinds: **constructors** build primitive objects without involving computation, and **operations** compute objects from other objects (including predicates, which we see as operations that return booleans). For purposes of interoperability it is desirable to abstract from this distinction and consider both as typed symbols. This abstraction is important because systems often disagree on the choice of constructors. Thus, we can represent the interfaces of the systems A to H as OMDoc/MMT theory graphs a to h that declare the constructors and operations (but omit all implementations of the operations) of the respective system.

Given the theory graph a representing the system dialect of A, we can express all objects in the language of system A as OMDoc/MMT objects using the symbols of a. We refer to these objects as A-objects. It is conceptually straightforward to write (or even automatically generate) the theory graph a and to implement a serializer and parser for A-objects as a part of A.[3] This is because no consideration of interoperability and thus no communication with the developers of other systems is needed.

[3]However, as we see below, this may still be surprisingly difficult in practice.

Alignments with the Ontology. The above reduces the interoperability problem to relating each system dialect to the MitM ontology. Each system dialect overlaps with the language of the ontology, but no system implements all ontology symbols and every system implements idiosyncratic operations that are not useful as a part of the ontology. Therefore, some system dialect symbols are related to corresponding symbols in the MitM ontology. We use these symbols of the MitM ontology as an intermediate representation to bridge between any two systems, e.g., by translating A-objects to the corresponding ontology objects and then those to the corresponding B-objects.

However, even when A and B deal with the "same mathematical objects", these may be constructed and represented differently, e.g., symbols can differ in name, argument order/number, types, etc. A major difficulty for system interoperability is correctly handling these subtle differences. To formalize the details of this relation, [Mül+17b] introduced OMDoc/MMT **alignments**. Technically, these are pairs of OMDoc/MMT symbol identifiers decorated by a set of key-value pairs. The alignments of a-symbols with the MitM ontology determine which A-objects correspond to MitM-objects.

The alignment of a-symbols to ontology symbols must be spelled out manually. But this is usually straightforward and easy even for inexperienced users. For example, the following line aligns GAP's symbol IsCyclic (in the file lib/grp.gd) with the corresponding symbol cyclic in the MitM ontology. The key-value pairs are used to signify that this alignment is part of a group of alignments called "VRE" and can be used for translations in both directions.

```
gap:/lib?grp?IsCyclic  mitm:/smglom/algebra?group?cylic
    direction="both" type="VRE"
```

Thus we can reduce the problem of interfacing n systems to *i*) curating the MitM ontology for the joint mathematical domain, *ii*) generating n theory graphs for the system dialects, *iii*) maintaining n collections of alignments with the MitM ontology.

Alignments form an independent part of the MitM interoperability infrastructure. Incidentally, they obey a separate development schedule: the MitM ontology is developed by the community as a whole as the understanding of a mathematical domain changes. The system dialects are released together with the systems according to their respective development cycle. The alignments bridge between them and have to mediate these cycles.

2.4. MitM-based Distributed Computation

The final missing piece for a system interoperability layer for a VRE toolkit is a practical way of transporting objects between systems. This requires two steps.

Firstly, if the system dialects and alignments are known, we can automatically translate A-objects to B-objects in two steps: A to ontology and ontology to B. This two-step translation has been implemented in [Mül+17a] based on the MMT system [Rab13; MMT], which implements the OMDoc/MMT format along with logical and knowledge management algorithms.

Secondly, each system A has to be able to serialize/parse A-objects and to send them to/receive them from MMT. In the OpenDreamKit project we use the OpenMath SCSCP (Symbolic Computation Software Composability) protocol [Fre+] for that. SCSCP is essentially a distributed remote-procedure-call system based on OpenMath, which is itself the fragment of OMDoc/MMT used for representing objects. It is straightforward to extend a parser/serializer for A-objects to an SCSCP clients/server by implementing the SCSCP protocol on top of, e.g., sockets or using an existing SCSCP library.

2.5. Mathematical Knowledge Bases

But the OpenDreamKit VRE must also include mathematical data sources, which curate the objects, models, examples, and counterexamples of mathematics. There are various large-scale sources of mathematical knowledge. These include

- generic information systems like 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 [Slo03; OEIS]), and the L-Functions and Modular Forms Database (LMFDB [Cre16; LMFDB]),
- fully formal theorem prover libraries like those of Mizar, Coq, PVS, and the HOL systems.

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 usually system specific, to use it, users 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. We claim that mathematicians and other scientists desire a "programmatic, mathematical API" that gives access to the knowledge-bases programmatically via their mathematical constructions and properties. We focus on addressing this problem in this paper.

For our implementation 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.

But knowledge 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:

- 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 it feeds.
- 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. Thus, from the system perspective, virtual theories behave just like concrete theories but without the assumption that all declarations are loaded at once; instead, declarations are loaded lazily.

We also update the knowledge management algorithms in the MMT system so that they can directly deal with the databases underlying the knowledge bases. Here we provide a systematic solution for encoding/decoding between low-level representations in standard databases and high-level mathematical representations.

3. The MitM Ontology

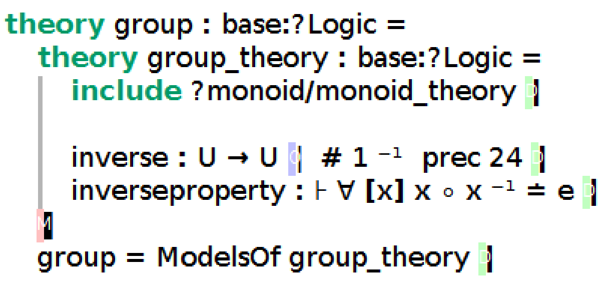
Jane's use case involves groups and actions, polynomials, rings and ideals, and Gröbner bases, all of which must be formalized in the MitM ontology. Due to space restrictions, we only describe the ontology for computational group theory (CGT) as an example. This formalization can be found at [Mitb].

3.1. Foundation

OMDoc/MMT formalizations must be relative to foundational logic, which is itself formalized in OMDoc/MMT. As foundation for all formalizations in MitM [Mita], we use a polymorphic dependently typed λ-calculus with two universes type and kind (roughly analogous to sets and proper classes in set theory) and subtyping. It provides dependent function types {a:A}B(a), representing the type of all functions mapping an argument a:A to some element of type B(a). If B does not depend on the argument a, we obtain the simple function type A→B.

For formulas, we use a type prop and a higher order logic where quantifiers range over any type. We furthermore follow the judgments-as-type paradigm by declaring a function ⊢:prop→type mapping propositions to the **type of their proofs**, which allows us to declare proof rules as functions mapping proofs (of the premises) to a proof (of the conclusion).

The judgment A<:B expresses that A is a subtype of B. We use power types (the type of subtypes of a type) and predicate subtyping {'a:A | P(a)'}. The latter makes type-checking undecidable, but that is necessary for natural formalizations in many areas of mathematics.



```
theory group : base:?Logic =
  theory group_theory : base:?Logic =
    include ?monoid/monoid_theory ❙

    inverse : U → U ❘ # 1 ⁻¹  prec 24 ❙
    inverseproperty : ⊦ ∀ [x] x ∘ x ⁻¹ ≐ e ❙
  ❚
  group = ModelsOf group_theory ❙
```

FIGURE 2. MitM Ontology Fragment

A critical question is what additional types to provide. To be practical at all, this must include basic types for aggregation (e.g., records) and collection (e.g., lists). Based on a survey of practical needs, we compiled such a set in Deliverable **D6.8.** [**ODK-D6.8**]

Moreover, we need an open-ended set of types for mathematical structures such as fields and rings. Here we have developed a novel type constructor [**MueRabKoh:tat18**] that turns any MMT theory into the corresponding dependent record type. This combines the benefits of MMT's axiomatic theories and a flexible type system. Concretely, for a theory T, the type ModelsOf T is the record type of models of T. For example, in Figure 2 we define groups in terms of an MMT theory group_theory that declares the operations and axioms in a way that corresponds to the mathematical definition of groups. Then group=ModelsOf group_theory is the type of all models of that theory, i.e., the type of groups. Any element g:group thus represents an actual group, whose operations and axioms can be accessed via record field projections (e.g. g.inverse yields the inverse operation of g. Since axioms are turned into record type fields as well, actually constructing a record of type group corresponds to proving that the field universe and the operations provided in the record do in fact form a group.

3.2. Domain Ontologies

Relative to the foundation, we can formalize the actual mathematical knowledge we are interested in. As a running example, we describe a formalization of computational group theory (CGT), which is inspired by the corresponding implementation in GAP. It uses several different levels of abstraction – currently *abstract*, *representation*, *implementation*, and *concrete*. From our experience, we expect this pattern to be applicable across computational algebra, possibly with additional levels of abstraction. The left box in Figure 3 shows the levels and their relation to the constructors and operations of GAP.

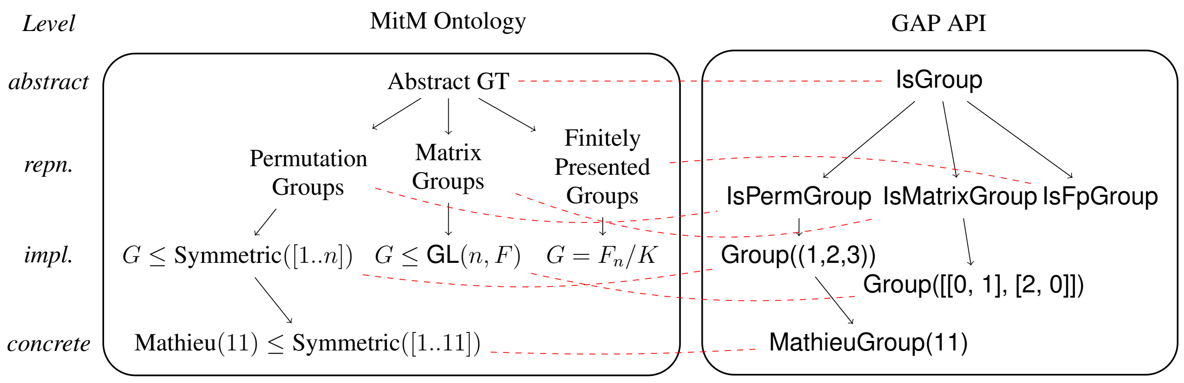


FIGURE 3. Alignments between the MitM Ontology and the GAP API

Abstract Level. This contains the theory of *Groups*: the group axioms, generating sets, homomorphisms, group actions, stabilisers, and orbits. This also easily leads into definitions of centralisers – i.e. stabilisers of elements under conjugation – and normalisers – i.e. stabilisers of subgroups under conjugation, stabiliser chains, Sylow-p subgroups, Hall subroups, and many other concepts.

OMDoc/MMT also allows expressing that there are different equivalent definitions of a concept: We defined group actions in two ways and used *views* to express their equivalence.

Representation Level. Abstract groups are represented in different ways as concrete objects suitable for computation: as groups of permutations, groups of matrices, finitely presented groups, algebraic constructions of groups, or using polycyclic presentations.

Many representations arise naturally from *group actions*: If we are considering symmetry in a setting where we want to apply group theory, we start with a group action, for example a group acting on a graph by permuting its vertices.

The universal tool to bridge the gap between groups, representations and canonical representatives are *group homomorphisms*, particularly embeddings and isomorphisms, which are used extensively in GAP. This is reflected in our approach.

Implementation Level. At this level we encode implementation details: Permutation groups in GAP are considered as finite subgroups of the group $S_{\mathbb{N}+}$, and defined by providing a set of generating permutations. GAP then computes a stabiliser chain for a group that was defined this way, and naturally considers the group to be a subgroup of $S_{[1..n]}$, where n is the largest point moved.

Concrete Level. It is at the concrete level where the computation happens: while the higher levels are suitable for mathematical deduction and inference, this level is where GAP (or any other system providing computational group theory) does its work. If a group (or a group action) has been constructed by giving generators through MitM, GAP can now compute the size of the group, its isomorphism type, and perform all the other operations that are available via the GAP system dialect.

[3] EdN:3

[3]EDNOTE: FR: the formalization of elliptic curves could be moved here as a second example

[4]

4. Concrete Encodings of MitM Objects

When integrating systems with the star-shaped MitM architecture, some translation of concrete formats is necessary. While this is not surprising, it leads to an important different between the integration of computation systems and databases: the former but not the latter include a programming environment that provides all necessary infrastructure for implementing the reformatting. Therefore, to integrate with databases, it is convenient to standardize some encodings that translate between high-level datatypes in the MitM ontology and concrete representations that can be send to and received from databases.

This is particularly critical as the databases used for the scalable physical storage of large datasets usually offer only very simple data structures. For example, a JSON database (as underlies LMFDB) offers only limited-precision integers, boolean, strings, lists, and records as primitive objects. An SQL database offers only records of basic objects. Neither provides 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. Consider, for example, the degree of an elliptic curve (as we will in Section 6. Its *semantic* type is $\mathbb{Z}$, but its *physical* type in LMFDB is IEEE 754 a mixture of 64-bit floating point numbers and strings: integers that exceeds $2^{53}-1$ are stored as JSON strings containing the corresponding decimal representation.

To formally specify these encodings codecs, we introduce a new OMDoc/MMT theory `Codecs` as a part of the MitM ontology. Our codecs are indexed by semantic types: the type constructor `codec` maps a semantic type to a new type of codecs for it. For instance, the object StandardInt of type `codec` $\mathbb{Z}$ is a codec that translates between LMFDB's idiosyncratic float/string-representation and MitM's integers. Note that there can be multiple different codecs for the same semantic type. For example, IntAsArray encodes integers x as lists of 64-bit integers consisting of the digits of x with respect to base 2^{64}.

Codecs			
codec	:	`type` $\to$ `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 4. Codecs specified in MMT ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}^+$ are as usual, $\mathbb{B}$ are booleans, and $\mathbb{S}$ are Unicode strings)

We do not (and do not have to) define the actual encoding/decoding functions in OMDoc/MMT. It is more important to identify the codecs needed in practice, introduce names for them, and spell out their semantics. Then it is straightforward to implement them in any other programming language used interfacing with LMFDB.

In particular, we have implemented them in Scala, the language underlying the MMT system. Additionally, the Codecs theory annotates each codec declaration with a reference to the Scala

[4]EDNOTE: FR: eventually we should move more content from list of types and the list of codecs here, which is in the D6.8 for now

class implementing the codec. That way, MMT can run the encoding/decoding functions of the codec.

The above is only sufficient for atomic semantic types, which typically correspond to one (or more) atomic codecs. Consider now the field isogeny_matrix of elliptic curves. The semantic representation of one possible value (namely for the curve 11a1) of this field is the matrix on the right.

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

The semantic type operator Matrix takes 1 type argument (the element type, integers in this case) and two value arguments (the dimensions, 3 and 3 in this case) and constructs the respective matrix type. In principle, one could give a codec for each matrix type that comes up in a database schema. But a much more elegant solution is to specify **codec operator**s in analogy to type operators. A codec operator for a type operator with k type and l value arguments, takes k codec and l value arguments. For example, a codec operator for matrices takes a codec $C : \texttt{codec}\, E$ for the element type E and the dimensions m and n and returns a codec of type $\texttt{codec}\,(\mathrm{Matrix}\, E\, m\, n)$.

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

FIGURE 5. Second annotated subset of the codecs theory containing a selection of codec operators found in MMT. Compare with Figure 4.

Like codecs, codec operators are represented in MMT in two ways: as declarations inside the theory Codecs (see Figure 5 for a list, compare again with Figure 7) and as a corresponding Scala function that maps codecs to codecs. When reading the declarations, note that we make use of the dependent function types of the MitM foundation: curly brackets denote dependent function arguments, i.e., arguments that may occur in later argument types and the result type.

With these declarations, we recover the LMFDB encoding of isogeny matrices by applying the codec operator StandardMatrix , which encodes matrices as lists of lists, to the codec StandardInt and the dimension 3 and 3. The resulting codec

$$\texttt{StandardMatrix}(\mathbb{Z}, 3, 3, \texttt{StandardInt})$$

encodes the above matrix as [[1,5,25],[5,1,5],[25,5,1]].

5. Integrating Computation Systems with MitM: GAP, SageMath, and Singular

We now show how we produce OMDoc/MMT theory graphs that specify the system dialects of GAP, Singular, and SageMath. The three systems are sufficiently different that we can consider the development presented in this section a meaningful case study in the methodology and difficulty of exposing the APIs of real-world systems as of formally described system dialects.

In each case, we had to overcome major implementation difficulties and invest significant manpower. In fact, even the serialization of internal abstract syntax trees as OMDoc/MMT objects proved difficult, for different system-specific reasons. In the following, we summarize these efforts.

5.1. SageMath

We first consider our previous work [Deh+16] regarding a direct (i.e., without MitM) integration of SageMath and GAP. Here SageMath's native interface to GAP is upgraded from the

handle paradigm to the **semantic handles** paradigm. In the former, when a system A delegates a calculation to a system B, the result r of the calculation is not converted to a native A object (unless it is of some basic type); instead B just returns a handle h (i.e., some kind of reference) to the B-object r. Later, A can run further calculations with r by passing it as argument to functions or methods implemented by B. Additionally, with a **semantic** handle, h behaves in A as if it was a native A object. In other words, one adapts the API satisfied by r in B to match the API for the same kind of objects in A. For example, the method call `h.cardinality()` on a SageMath handle `h` to a GAP group `G` triggers in GAP the corresponding function call `Size(G)`.

This approach avoids the overhead of back and forth conversions between A and B and enables the manipulation of B-objects from A even if they have no native representation in A. However, if these B-objects need to be acted on by native operations of A or other systems (as in Jane's scenario), we actually have to convert the objects r between A and B.

5.1.1. *API.* In [Deh+16] we describe the extraction of some of SageMath's API from its **categories**. This exploited the mathematical knowledge explicitly embedded in the code to cover a fairly large area of mathematics (hundreds of kinds of algebraic structures such as groups, algebras, fields, ...), with little additional efforts or need to curate the output. This extraction did not cover the constructors, knowledge about which is critical for (de)serialization, nor other areas of mathematics (graph theory, elliptic curves, ...) where SageMath developers currently do not use categories (usually because the involved hierarchies of abstract classes are shallow and easily maintained by hand).

To extract more APIs, we took the following approach:

(1) We constructed a list of typical SageMath objects.
(2) We used introspection to analyze those objects, crawling recursively through their hierarchy of classes to extract constructors and available methods together with some mathematical knowledge.

At this stage, the list of objects was crafted by hand to cover Jane's scenarios and some others. In a later stage, we plan to take advantage of one of SageMath's coding standards: every concrete type must be instantiated at least once in SageMath's tests and the instance passed trough a generic test suite that runs sanity checks for its advertised properties (e.g. associativity, ...). Therefore, by a simple instrumentation of SageMath's test framework, we could run our exporter on a fairly complete collection of SageMath objects.

The process remains brittle and the export will eventually require much curation:

- The signature of methods is incomplete: it specifies the number and names of the arguments, but only the type of the first argument.
- For constructors, the type of all the arguments is known, but only for the specific call that led to the construction of the introspected object.
- There is no distinction between mathematically relevant methods and purely technical ones like data structure manipulation helpers.
- The export is very large and seems of limited use without alignments with the MitM ontology. At this stage we do not foresee much opportunities to produce such alignments other than manually.

Nonetheless, we consider this an important first step toward fully automatic extraction of the SageMath API. Moreover, we expect further improvements by code annotations in SageMath (e.g., the ongoing porting of SageMath from Python 2 to Python 3 will enable **gradual typing**, which we hope to become widely adopted by the community) or using type inference in SageMath and/or MitM.

5.1.2. *Serialization and Deserialization.* Because SageMath is based on Python, it benefits from its native serialization support. For example, the dihedral group D_4 is serialized as a binary string, which encodes the following straight line program to be executed upon deserialization:

```
pg_unreduce = unpickle_global('sage.structure.unique_representation', 'unreduce')
```

```
  pg_DihedralGroup =
        unpickle_global('sage.groups.perm_gps.permgroup_named', 'DihedralGroup')
  pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
  pg_unreduce(pg_DihedralGroup, (pg_make_integer('4'),), {})
```

The first three lines recover the constructors for integers and for dihedral groups from SageMath's library. The last line applies them to construct successively the integer 4 and D_4.

Up to concrete syntax, this serialization is already close to the desired SageMath system dialect. We can therefore extend Python's native (de)serializer to use OMDoc/MMT as an alternative serialization format (using the Python library [POMa]). The following shows the corresponding OpenMath syntax tree in Python and XML respectively:

```
OMApplication(
  elem=OMSymbol(name='DihedralGroup',
                    cd='sage.groups.perm_gps.permgroup_named', cdbase='http://python.org'),
  arguments=[OMApplication(
    elem=OMSymbol(name='make_integer',
                      cd='sage.rings.integer', cdbase='http://python.org'),
    arguments=[OMBytes(bytes='4')])])
```

```
<OMA xmlns="http://www.openmath.org/OpenMath">
  <OMS name="DihedralGroup"
        cd="sage.groups.perm_gps.permgroup_named" cdbase="http://python.org"/>
  <OMA>
    <OMS name="make_integer" cd="sage.rings.integer" cdbase="http://python.org"/>
    <OMB>NA==</OMB>
  </OMA>
</OMA>
```

This approach has the additional advantage of benefiting from future optimizations implemented in Python's serialization, like structure sharing for identical subexpressions.

Still, systematically expanding OMDoc/MMT serialization to the *entire* SageMath library requires significant manpower and can only be a long-term goal. To increase community support, our design elegantly decouples the problem into (i) instrumenting the serialization to generate OMDoc/MMT as an alternative target format, and (ii) structural improvements of the serialization that benefit SageMath in general.

In particular, our serialization of SageMath objects is **by construction** rather than **by representation**, i.e., we serialize the constructor call that was used to build an object instead of the low-level Python representation of the resulting object. This is important to hide implementation details and allow for straightforward alignments. From the origin, the SageMath community has internally promoted good support for serialization as this is a fundamental building block for communication between parallel processes, databases, etc. Thus, it already values serialization by construction as superior because it is usually more concise and more robust under changes to SageMath. Therefore, independent of the purposes of this report, we expect a synergy with the SageMath community toward improving serialization.

5.2. GAP

In [Deh+16], we already described our general approach to extract APIs from the GAP system. We have now improved on this work considerably.

Firstly, we improved the MitM foundation so that the primitives of GAP's type system can be expressed in the MitM ontology.[4] GAP's type system heavily uses subtyping: **filters** express finer and finer subtypes of the universal type IsObject. Moreover, an object in GAP can learn about its properties, meaning its type is refined at runtime: a group can learn that it is Abelian or nilpotent and change its type accordingly.

Secondly, we devised and implemented a special treatment of GAP's constructors during serialization. As GAP only has a weak notion of object construction, we achieved this by manually identifying and annotating all functions that create objects in the GAP code base and then instrumenting them to store which arguments they were called with. With the constructor annotation in place, it is possible to have GAP represent any object in a running session as either a primitive type (integers, permutations, transformations, lists, floats, strings), or as a constructor applied to a list of arguments.

The instrumentation itself is minimal – 57 lines of GAP code, plus 100 lines for serializing and parsing. The main – and indeed considerable – challenge was to identify the constructors and their arguments. In GAP, objects are created by calling the function Objectify with a type and some arguments. Hence we analyzed all call-sites to this function and some light inference of the enclosing function. This amounted to 665 call sites in the GAP library and an additional 1664 in the standard package distribution. The instrumentation will be released as part of a future version of GAP, making GAP fully MitM capable. [5]

As a major positive side-effect of our work, this instrumentation led to general improvements of the type infrastructure in GAP. For example, it enables static type analysis, which can be used to optimize the dynamic method dispatch and thus hopefully lead to efficiency gains in the system.

5.3. Singular

As we only need a very small part of Singular for our case study, we were able to use the existing OpenMath content dictionaries for polynomials [OMCP] as the Singular system dialect. These are part of a standard group of content dictionaries that describe (some) mathematical objects at a high level of abstraction to be universally applicable. OMDoc/MMT understands OpenMath, i.e., it can use these content dictionaries as OMDoc/MMT theories.

Building on the OpenMath toolkits for OpenMath phrasebooks [POMa] and SCSCP communication [POMb] in Python – which were developed for SageMath in the OpenDreamKit project, we wrapped Singular in a thin layer of Python code that provides SCSCP communication. This work was undertaken by the sixth author as part of a summer internship in about a week without prior expert knowledge of the system. Of course, if we want to achieve a more comprehensive coverage of the Singular dialect, we will have to either manually write a theory graph or instrument Singular for extraction as we did for SageMath or GAP above.

5.4. Alignments

Finally we have to curate the alignments between the system dialects and the MitM ontology. These alignments are currently produced and curated manually using the approach, repository, and syntax described in [Mül+17b; Mül+17a]. In the future, we will also consider automatically extracting alignments from the existing ad-hoc SageMath-to-X translations. These are (mainly) given as SageMath code annotations that relate SageMath operations and constructors with those of system X.
[6]

[4]In the future MMT might even serve as an external type-checker for GAP.

[5]EdNote: MP: Put an example of OM_Print here, maybe for a group, or for Cosets (as they are something that the standard OpenMath CDs in GAP cannot do)

[6]EdNote: FR: This section is very weak. It should give examples of alignments and discuss some of the difficulties and solutions.

6. Integrating Databases with MitM: LMFDB

The mathematical software systems to be integrated via the MitM approach have so far been computation-oriented, e.g., computer algebra systems. Their API theories 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 mathematical database defines

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

Thus, we can reuse many of the same concepts. In particular, the API theories must contain definitions of the database schemas.

Apart from standard software engineering tasks, this leaves three conceptual problems we had to solve:

P1 Turn the database schemas and tables into OMDoc/MMT theories and declarations.

P2 Lift data in *physical* representation (as records of the underlying database) to OMDoc/MMT object in *semantic* representation.

P3 Translate semantic queries to queries about physical representations so that they can be executed directly on the database without loading the entire theory into MMT.

We deal with **P1** in Section 6.2, with **P2** in Section 6.3, and with **P3** in Section 6.4.

In this paper, we focus on LMFDB as an example, of which we give an overview in Section 6.1. However, our methods are general enough to apply to many other mathematical databases such as OEIS, or findstat.

6.1. LMFDB Overview

The "L-Functions and Modular Forms Database" (LMFDB [LMFa]) is a large database, storing among other mathematical objects millions of L-Functions, modular forms and curves, along with their properties. Technically, it uses a MongoDB[5] database with a Python web frontend.

LMFDB has several sub-databases, e.g., for elliptic curves or transitive groups. Within each of these, every object is stored as a single JSON record. Figure 6 shows an example: 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 modular parametrization of the underlying elliptic curve.

```
{
    "degree": 1,
    "x-coordinates_of_integral_points": "[5,16]",
    "isogeny_matrix": [[1,5,25],[5,1,5],[25,5,1]],
    "label": "11a1",
    "_id": "ObjectId('4f71d4304d47869291435e6e')",
    ...
}
```

FIGURE 6. Part of an elliptic curve in LMFDB (some fields omitted for brevity)

Other properties are more complex: the value of the isogeny_matrix property is a list of lists representing a matrix. This disconnect between JSON encoding and mathematical meaning can become much more severe, e.g., the x-coordinates_of_integral_points field is semantically a list of integers but (due to the sizes limits on integers) is encoded as a string.

The LMFDB API [Lmf] exposes a querying interface that can be used either by humans via the web or programmatically via JSON-based GET requests over HTTP. Queries must name the sub-database to be queried and consist of a set of key-value pairs that correspond to an SQL

[5]to be replaced by PostgreSQL in 2018

`where` clause. However, while LMFDB offers a programmable API for accessing its contents, this API sits at the level of the underlying MongoDB, and not the level of mathematical objects. For example, to retrieve all Abelian objects in the subdatabase of transitive groups, we expect to use the key-value pair commutative $=$ true . However, these values need to be encoded to be understood by MongoDB. We need to realize that the database schema actually uses the key ab for commutativity, that it has boolean values, and that the schema encodes `true` as `1`. Thus, the actual query to send is `http://www.lmfdb.org/api/transitivegroups/groups/?ab=1`.

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

6.2. LMFDB as a Set of Virtual Theories

The set of constants in a database table – while finite – can be arbitrarily large. In particular, all LMFDB tables[6] 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 as 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 practical to fix a set of concrete API theories. Instead, the API theories must be 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).

It is straightforward how to implement each LMFDB table as a virtual MMT theory V: we use an initially empty concrete theory C, and whenever an identifier id of V is requested, MMT dynamically adds the corresponding declaration of id to C. Because MMT already abstracts from the physical realizations of persistent storage, we only have to implement a new storage instance that connects to LMFDB, retrieves the JSON object with identifier id, and turns it into an OMDoc/MMT declaration.

A sketch of our overall solution is given in Figure 7. The relevant parts of the MitM ontology comprise simple ones like numbers and matrices (in green) and LMFDB-specific ones like elliptic curves (in red). The remaining theories (in blue) form the LMFDB API theories: the schema theory and the database theory, which we describe below.

LMFDB's original technical realization, using MongoDB, did not require formalizing the schema of each table. Instead, the tables were generated systematically and therefore followed an implicit schema that could — in principle — be obtained from the documentation or reverse-engineered from the tables. Until 2017 the documentation of these implicit schemas was created and maintained manually by LMFDB developers, and as a result was incomplete and frequently

[6]Technically, until July 2018, LMFDB was implemented using MongoDB and comprises a set of sets (each one called a database) of JSON objects. MongoDB allows each JSON object in a collection to be different (with a different schema), though in practice almost all objects in each collection had the same schema apart from some missing data components. The schema for each collection had to be documented elsewhere, in an inventory, which since 2017 has been itself stored as a database within the LMFDB. During 2018, however, work has been ongoing to migrate the LMFDB to use PostgreSQL (with a fixed schema for each table) as the underlying database, without any change to the external API. In both cases, due to the conventions used, we can understand the LMFDB conceptually as a set of tables of a relational database, keeping in mind that every row is a tuple of arbitrary JSON objects.

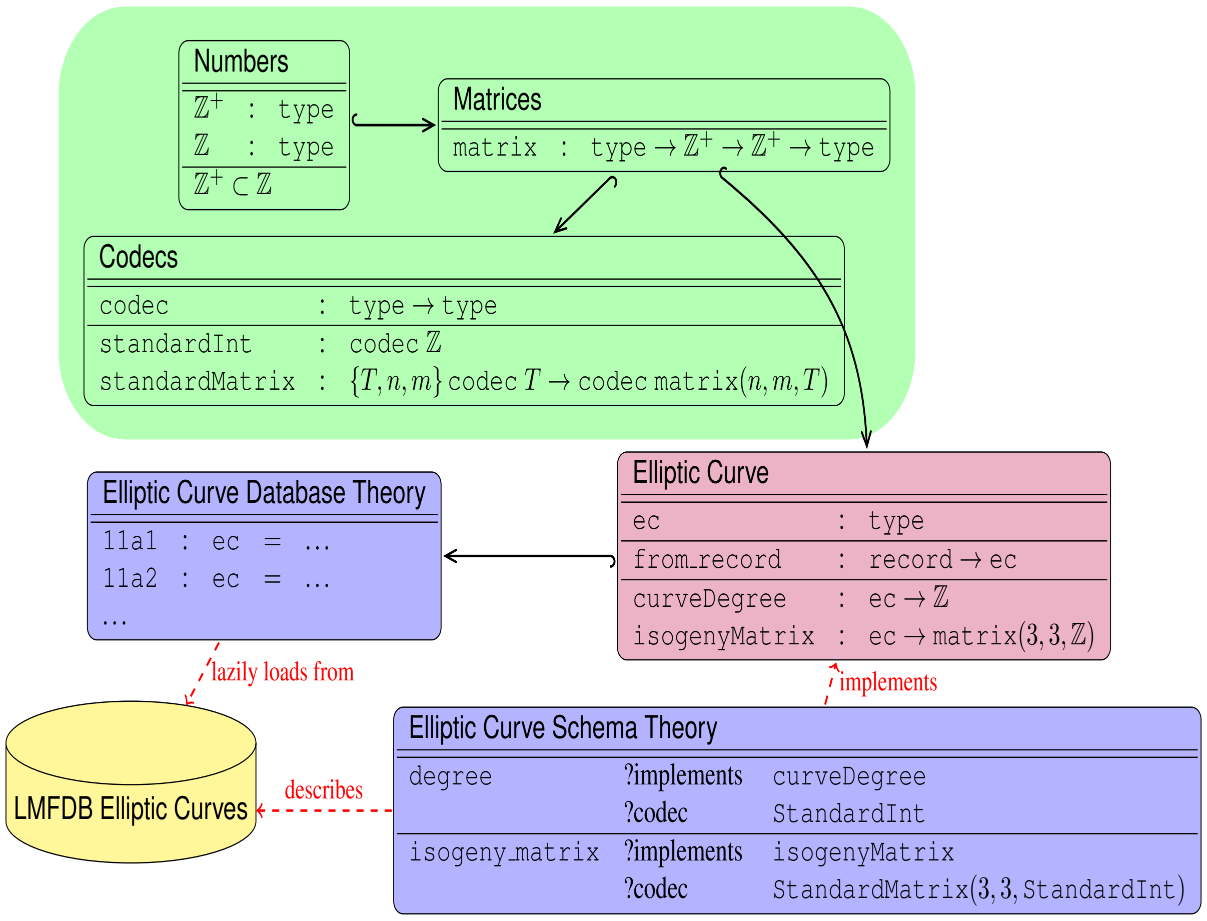

Numbers
$\mathbb{Z}^+$: type
$\mathbb{Z}$: type
$\mathbb{Z}^+ \subset \mathbb{Z}$
Matrices
matrix : type → $\mathbb{Z}^+$ → $\mathbb{Z}^+$ → type
Codecs
codec : type → type
standardInt : codec $\mathbb{Z}$
standardMatrix : $\{T,n,m\}$ codec T → codec matrix(n,m,T)
Elliptic Curve Database Theory
11a1 : ec = ...
11a2 : ec = ...
...
Elliptic Curve
ec : type
from_record : record → ec
curveDegree : ec → $\mathbb{Z}$
isogenyMatrix : ec → matrix$(3,3,\mathbb{Z})$
lazily loads from
implements
LMFDB Elliptic Curves
describes
Elliptic Curve Schema Theory
degree ?implements curveDegree
?codec StandardInt
isogeny_matrix ?implements isogenyMatrix
?codec StandardMatrix(3,3,StandardInt)

FIGURE 7. Virtual theory for LMFDB elliptic curves (some declarations omitted)

out of date. During 2017-2018 a new LMFDB Inventory was created, taking as its starting point both the manually prepared inventory (which contained human definitions and explanations of the content of each data field) and a new dynamically created schema obtained by analysis of the data in each collection. This process, which was both necessitated by the requirements of the Math-in-the-Middle approach and made possible in practice through the provision of research software engineers funded by ODK, revealed numerous inconsistencies in the LMFDB data which developers have since been able to address. Moreover, having this detailed schema for each collection in the MongoDB databases also fed in to the migration process, expected to be complete by August 2018, in which the MongoDB free-format collections are being replaced by PostgreSQL tables, each of which has a completely specified and formalized schema.

Note that (and here LMFDB critically differs from, e.g., the OEIS), the mathematical definitions and concepts involved in the LMFDB data and tables are extremely deep so that reverse-engineering the associated schemas from the data itself is only possible in practice with the aid of experts. As the first such table for which a formal schema was to be created, before the development of the new comprehensive Inventory, we chose one for which the existing documentation was most complete, and which originated with one of the current authors (John Cremona), who sat down with the Math-in-the-Middle team at an ODK workshop to formalize 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.

Our formalization models elliptic curves in a very simple fashion by using an abstract type ec . The constructor from_record takes an MMT record and returns an elliptic curve. Properties of elliptic curves are formalized as functions out of this type. We list only two here as examples: the degree, an integer, and the isogeny matrix, a 3×3 matrix of integers. We omit the relevant axioms, which are not essential for our purposes here. As usual for the MitM approach, 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.

6.3. Ascribing Encodings in Schema Theories

If we ignore encoding issues, schema theories are straightforward: they contain one declaration of the same name for each field within an LMFDB record. This specifies only the semantic type of each field and does not relate it to the MitM formalization. To handle the encoding as a physical type, we annotate each declaration with the codec that the databases for the values of that field. Moreover, to connect the schema theories to the MitM formalization, we additionally annotate each field with the corresponding property of elliptic curves from the MitM theory. We can now understand the last unexplained parts of Fig. 7. ?implements is the symbol used to annotate the metadatum, which MitM property a schema field corresponds to. And ?codec similarly annotates the codec to each field.

For example, the degree field implements the curveDegree property in the elliptic curve theory and uses the StandardInt codec. Thus, the schema theories determine the entire relation between semantic and physical objects.

The database theory is a virtual theory and contains one declaration per LMFDB record. Given the URI of an object in the respective database, our MMT backend for LMFDB first retrieves the appropriate record from LMFDB – in the case of 11a1 this corresponds to retrieving the JSON found in Figure 6. Then, for each field, it uses the annotated codec (which is an OMDoc/MMT expression) to build an actual codec (as a runnable Scala function) and runs its decoding function. Next, it passes the resulting record to the from_record constructor, which yields an elliptic curve in the MitM theories. Finally, this elliptic curve is added as a new declaration in the database theory.

6.4. Translating Queries

Recall that MMT has a general-purpose Query Language called QMT [Rab12], which allows users to find knowledge subject to even complex conditions. We continue by briefly addressing **P3**: query translation; 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 has a specific value. As an example, consider again the query of finding all Abelian transitive groups. QMT has an MMT-powered surface syntax, which can be used to express this query as:

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

The example consists of two parts, first we find all objects declared in the theory interface theory lmfdb:db/transitivegroups?group (line 1), and then we restrict this set of results to all those for which the `commutative` property is `true` (line 2). Notice that this the example shown here is the formal equivalent of the LMFDB query shown in Section 6.1. The key difference is that this query does not require knowing the record structure of LMFDB– apart from knowing the proper sub-db, instead it only relies on knowing the mathematical semantics (commutativity) of the query in question.

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 had been produced. But it is infeasible 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.

The low-level API of LMFDB and similar system 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 currently 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. However, it is insufficient to simply translate queries as a whole: 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, and assign them to a specific backend, and then translate only these parts.

Achieving this automatically is a non-trivial problem. Queries are inductive in nature, and 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 computationally expensive implementation.

Instead of intercepting each result, we extended the Query Language to allow users to annotate sub-queries for evaluation with a specific virtual theory backend. This allows the system to immediately know which parts of a query have to be evaluated in MMT memory, and which have to be translated and sent to an external system. This turns the example above into:

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

Here, we have simply wrapped the entire query with a `use lmfdb` statement, indicating the query should be evaluated using LMFDB.

The encoding of this specific query can be achieved using codecs. The query corresponds to the URL http://www.lmfdb.org/api/transitivegroups/groups/?ab=1. Next, the LMFDB API returns a set of JSON objects corresponding to all Abelian transitive groups. These can then be decoded into OMDoc/MMT objects using the procedure described in Section 6.3, i.e. for each field we look up the corresponding codec and use it to deconstruct the field, eventually creating an MMT record. Afterwards, these OMDoc/MMT terms can then be passed to the user as a result to the query.

7. MitM-Based Distributed Computation

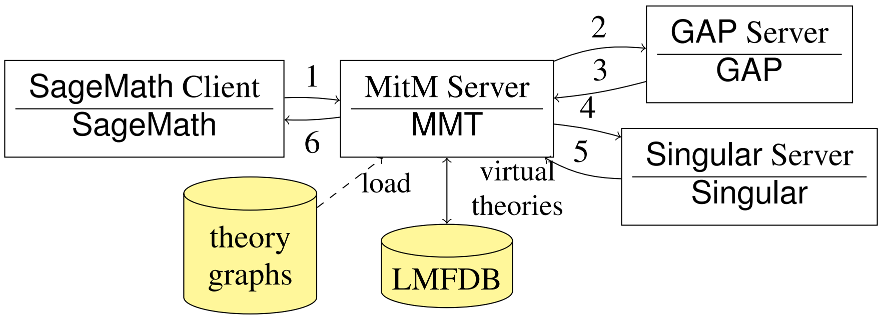


FIGURE 8. MitM Interaction in Jane's Use Case

Figure 8 shows the overall architecture with an MitM server as the central mediator. All arrows represent the transfer of OMDoc/MMT objects via SCSCP. Critically, the MitM server also maintains the alignments and uses them to convert between system dialects.

We have extended the MMT system [Rab13] with an SCSCP server/client so that it can receive/send objects from/to computation systems. For the GAP server, we built on pre-existing SCSCP support. To obtain an SCSCP server for Singular, which does not have native SCSCP support, we wrapped Singular in a python script that includes the pyscscp library [POMb]. In SageMath, we directly programmed the client interface to the MitM server.

The resulting system forms the nucleus of the OpenDreamKit interoperability layer. It can already delegate computations between the three participating systems as long as the exchanged objects are covered by the MitM ontology, the alignments, and the formalizations of the system dialects.

Jane's Use Case. Initially, Jane has already built in SageMath the ring $R = \mathbb{Z}[X_1, X_2, X_3, X_4]$, the group $G = D_4$, the action A of G on R that permutes the variables, and the polynomial $p = 3 \cdot X_1 + 2 \cdot X_2$. She now calls

```
MitM.Singular(MitM.Gap.orbit(G, A, p)).Ideal().Groebner().sage()
```

which results in the following steps (the numbers on the edges of the graph of Figure 8 indicate the order of communications when processing Jane's use case):

(1) Jane uses SageMath to call the MitM server with the command above, which includes both the computation to be performed and information about which system to use at which step.
(2) The MitM server translates MitM.Gap.orbit(G, A, p) to the GAP system dialect and sends it to GAP.
(3) GAP returns the orbit:

$$\begin{aligned} O = [&3X_1 + 2X_2, 2X_3 + 3X_4, 3X_2 + 2X_3, 3X_3 + 2X_4, \\ &2X_2 + 3X_3, 3X_1 + 2X_4, 2X_1 + 3X_4, 2X_1 + 3X_2] \,. \end{aligned}$$

(4) The MitM server translates MitM.Singular(O).Ideal().Groebner() to the Singular system dialect and sends it to Singular.
(5) Singular returns the Gröbner base B.
(6) The MitM server translates B to the SageMath system dialect and sends it to SageMath, where the result is shown to Jane.

$$B = [X_1 - X_4, X_2 - X_4, X_3 - X_4, 5X_4] \,.$$

Alternative Use Case. Suppose Jon, one of Jane's colleagues, prefers working in GAP, and he wants to compute the Galois group of the rational polynomial $p = x^5 - 2$. He discovers the GAP package `radiroot`, which promises this functionality, but unfortunately the package does not work for this polynomial and thus GAP alone cannot solve Jon's problem.

Jon hears from Jane that he should use SageMath, because she knows it can compute Galois groups. So, from GAP, he calls

```
G := MitM("Sage", "GaloisGroup",p)
```

which gives him the desired Galois group as a GAP permutation group. Having heard of Jane's experiments, he can further run her orbit and Gröbner basis calculation starting from this new group, without leaving his favorite computing environment.

Finally, Jon, being a proficient GAP user, also knows that he can now install a **method** in GAP by calling

```
InstallMethod(GaloisGroup, "for a polynomial", [IsUnivariatePolynomial],
                    p -> MitM("Sage", "GaloisGroup", p))
```

that will compute the Galois group of any rational polynomial transparently for him whenever he calls GaloisGroup for a rational polynomial in GAP. And thus (at the price of using multiple systems) a significant part of the 1800-line radiroot package can be replaced by a few lines in GAP, taking advantage of the work of the SageMath community and participating in any future improvements of SageMath. In fact, Sage itself delegates to the PARI system – another one of the OpenDreamKit systems – for this computation. So in the future GAP might directly delegate to PARI instead, bypassing the need of iterated translations.

[7] EdN:7

[8] EdN:8

8. Conclusion

We have implemented the MitM approach to integrating mathematical software systems based on formalizations of the underlying mathematical knowledge. The main investment here was the curation of an MitM Ontology, the generation of formal specifications of system APIs for SageMath, GAP, and Singular, identifying the alignments of these APIs with the ontology, implementing an MitM server that can use alignments to translate between systems, and implementing the SCSCP protocol for all involved systems.

We have also shown how to extend the Math-in-the-Middle framework for integrating systems to mathematical data bases like the LMFDB. The main idea is to embed knowledge sources as virtual theories, i.e. theories that are not – theoretically or in practice – limited in the number of declarations and allow dynamic loading and processing. For accessing real-world knowledge sources, we have developed the notion of codecs and integrated them into the MitM ontology framework. These codecs (and their MitM types) lift knowledge source access to the MitM level and thus enable object-level interoperability and allow humans (mathematicians) access using the concepts they are familiar with. Finally, we have shown a prototypical query translation facility that allows to delegate some of the processing to the underlying knowledge source and thus avoid thrashing of virtual theories.

Related Work. Most other integration schemes employ a **homogenous approach**, where there is a master system and all data is converted into that system. A paradigmatic example of this is the Wolfram Language [Wik17] and the Wolfram Alpha search engine [Wol], which are based on the Mathematica kernel. This is very flexible for anyone owning a Mathematica license and experienced in the Mathematica language and environment.

The MitM-based approach to interoperability of data sources and systems proposed in this paper is inherently a **heterogeneous approach**: systems and data sources are kept “as is”, but their APIs are documented in a machine-actionable way that can be utilized for remote procedure calls, content format mediation, and service discovery. As a consequence, interaction between systems is very flexible. For the data source integration via virtual theories presented in this paper this is important. For instance, we can just make an extension of MMT or SageMath which just act as a programmatic interface for e.g. LMFDB.

Our case studies show that MitM-based integration is an achievable goal. Delegation-based workflows can either be programmed directly or embedded into the interaction language of the mathematical software systems.

The main advantages and challenges claimed by the MitM framework come from its loosely coupled and knowledge-based nature. Compared to ad-hoc translations, MitM-based interoperability is relatively expensive as objects have to be serialized into (possibly large) OMDoc/MMT

[7]EDNOTE: $p = x^4 - x^3 - x^2 + x + 1$ over $\mathbb{Q}$ would have D_8 as galois group again...

[8]EDNOTE: FR: an old vision of a use case is commented out here; parts of it may still be useful

objects, transferred via SCSCP to MMT, parsed, translated into another system dialect, serialized and transferred, and parsed again. On the other hand, instead of implementing and maintaining n^2 translations, we only have to establish and maintain n collections of system APIs and their alignments to the MitM ontology. This makes the management of interoperability much more tractable:

(1) The MitM ontology is developed and maintained as a shared resource by the community. We expect it to be well-maintained, since it can directly be used as a documentation of the functionality of the respective systems.
(2) All the workflows are star-shaped: instead of requiring expert knowledge in two systems – a rare commodity even in open-source projects, and even for the system experts involved in this report– and keeping up with their changes, the MitM approach only needs expertise and change management for single systems.

All in all, these translate into a "business model" for MitM-based cooperation in terms of the necessary investment and achievable results, which is based on the well-known *network effects*: the joining costs are in the size of the respective system, whereas the rewards – i.e. the functionality available by delegation – is in the size of the network.

This network effect can be enhanced by technical refinements we are currently studying: For instance, if we annotate alignments with a "priority" value that specifies how canonically/efficiently/powerfully a given system implements a given MitM operation, then we can let the MMT mediator automatically choose a suitable target system for a requested computation (as opposed to our current setup where Jane specifies which systems she wants to use). On the other hand, for workflows where we do not need or want service-discovery, alignments can be "compiled" into n^2 transport-efficient direct translations that may even eliminate the need for serialization and parsing.[9]

We have shown how to extend the Math-in-the-Middle framework for integrating systems to mathematical data bases like the LMFDB. The main idea is to embed knowledge sources as virtual theories, i.e. theories that are not – theoretically or in practice – limited in the number of declarations and allow dynamic loading and processing. For accessing real-world knowledge sources, we have developed the notion of codecs and integrated them into the MitM ontology framework. These codecs (and their MitM types) lift knowledge source access to the MitM level and thus enable object-level interoperability and allow humans (mathematicians) access using the concepts they are familiar with. Finally, we have shown a prototypical query translation facility that allows to delegate some of the processing to the underlying knowledge source and thus avoid thrashing of virtual theories.[9]

Related Work. Most other integration schemes employ a **homogenous approach**, where there is a master system and all data is converted into that system. A paradigmatic example of this is the Wolfram Language [Wik17] and the Wolfram Alpha search engine [Wol], which are based on the Mathematica kernel. This is very flexible for anyone owning a Mathematica license and experienced in the Mathematica language and environment.

The MitM-based approach to interoperability of data sources and systems proposed in this paper is inherently a **heterogeneous approach**: systems and data sources are kept "as is", but their APIs are documented in a machine-actionable way that can be utilized for remote procedure calls, content format mediation, and service discovery. As a consequence, interaction between systems is very flexible. For the data source integration via virtual theories presented in this paper this is important. For instance, we can just make an extension of MMT or Sage which just act as a programmatic interface for e.g. LMFDB.

[9] OLD PART: MK: the MACIS-17-interop conclusion

Future Work. [10] [11] We have discussed the MitM+virtual theories methodology on the elliptic curves sub-base of the LMFDB, which we have fully integrated. We are currently working on additional LMFDB sub-bases. The main problem to be solved is to elicit the information for the respective schema theories from the LMFDB community. Once that is accomplished, specifying them in the format discussed in this paper and writing the respective codecs is straightforward. EdN:10 EdN:11

Moreover, we are working on integrating the the Online Encyclopedia of Integer Sequences (OEIS [Slo03; OEIS]). Here we have a different problem: the OEIS database is essentially a flat ASCII file with different slots (for initial segments of the sequences, references, comments, and formulae); all minimally marked up ASCII art. In [LK16] we have already (heuristically) flexiformalized OEIS contents in OMDoc/MMT; the next step will be to come up with codecs based on this basis and develop schema theories for OEIS.

Acknowledgments. The authors gratefully acknowledge the fruitful discussions with other participants of work package WP6, in particular Alexander Konovalov on SCSCP, Paul Dehaye on the SageMath export and the organization of the MitM ontology, Luca de Feo on OpenMath phrasebooks and the SCSCP library in python.

We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541) and DFG project RA-18723-1 OAF.

[10] EDNOTE: MK: this is essentially future work only for LMFDB, we need future work also for MitM here.

[11] EDNOTE: MK@FR: could you please describe the MMT Python bridge and how this could be used for SCSCP-less communication with Sage. The MitM-work here is to allow for compilation of the alignment based translations into code.