REPORT ON OpenDreamKit DELIVERABLES

D6.2: INITIAL $\mathcal{DKS}$ BASE DESIGN (INCLUDING BASE SURVEY AND REQUIREMENTS WORKSHOP REPORT)

D6.3: DESIGN OF TRIFORM ($\mathcal{DKS}$) THEORIES (SPECIFICATION/RNC SCHEMA/EXAMPLES) AND IMPLEMENTATION OF TRIFORM THEORIES IN THE MMT API

PAUL-OLIVIER DEHAYE, MIHNEA IANCU, MICHAEL KOHLHASE, ALEXANDER KONOVALOV, SAMUEL LELIÈVRE, DENNIS MÜLLER, MARKUS PFEIFFER, FLORIAN RABE, NICOLAS M. THIÉRY, AND TOM WIESING

Due on	31/08/2016 (Month 12) and 30/11/2016 (Month 15)
Delivered on	12/09/2016
Lead	Jacobs University Bremen (JacobsUni)
Progress on and finalization of this deliverable has been tracked publicly at: https://github.com/OpenDreamKit/OpenDreamKit/issues/136	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #136 ON 2016-09-19

- **WP6:** Data/Knowledge/Software-Bases
- **Lead Institution:** Jacobs University Bremen
- **Due:** 2016-08-31 (month 12)
- **Delivered:** 2016-09-12
- **Nature:** Report
- **Tasks:** T6.1 (#123), T6.3 (#125), T6.5 (#127)
- **Proposal:** p.55
- **Final report**

The OpenDreamKit proposal had envisioned WP6: *Data/Knowledge/Software bases* as a foundational enterprise that would develop a knowledge-based architecture over the course of the project and would allow to re-engineer *ad-hoc* interfaces between systems (e.g. from T3.2 (#51)) on a more *semantic* basis – the knowledge aspect (K). Consequently, the proposal envisioned concentrating the data (D) aspect on the mathematical knowledge bases (specifically LMFDB, OEIS, and FindStat) and proposed a host of foundational investigations of mathematical for the software (S) aspect with applications e.g. in the verification of algorithms.

Already the kickoff meeting in Paris in September 2015 revealed that the D/K/S aspects are much more tightly coupled in systems than anticipated. This was confirmed by the DKS survey conducted subsequently. In particular, the participants of WP6 identified the interoperability of OpenDreamKit systems to be one of the most critical steps in creating a VRE toolkit. Thus we prioritized tasks T6.1 (#123), T6.2 (#124), T6.3 (#125) and organized a series of workshops and code-maratons to develop a semantic foundation for system interoperability and simultaneously test it in implementations.

As a consequence, we have completed the initial design of D/K/S-bases (for this deliverable) in parallel with the initial implementation of a DKS base format based on OMDoc/MMT and the implementation of a DKS base system itself based on the MMT system (both for D6.3 (#137)), all activities fuelling each other. D6.3 (#137) was thus completed about three months ahead of schedule. Note that the RNC schema envisioned in the title proved un-necessary since, with the refined Math-in-the-Middle (MitM) design, the normal OMDoc/MMT schema is sufficient. Due to the resulting tight coupling between this deliverable and D6.3 (#137), and for the convenience of the reader, we have decided to report on both deliverables together.

In this report we therefore present the design process towards *DKS*-theories including the overall architecture (for this deliverable), a survey of the systems involved (for D6.3 (#137)), our current implementation (for D6.3 (#137)) as well as our plans for the future. Each part is labeled by the deliverable they contribute to mainly.

CONTENTS

1. Introduction

The goal of the OpenDreamKit project is to develop a generic toolkit that will enable mathematicians (and scientists in general) to build so-called Virtual Research Environments (VREs) that are optimally tailored to specific communities. These will combine a multitude of different tasks, such as symbolic mathematics, automatic code generation, numerical computation, data bases, post-processing or visualisation. A VRE will provide end-users with a single tool-chain that can be used for most, if not all, of their research.

To be able to build such a toolkit, we will need to combine three different aspects of "doing mathematics research" – Data (D), Knowledge (K) and Software (S). Ultimately we want to create and make use of these through a VRE: we want to model the real world, translate it into a set of mathematical objects and computationally simulate and thereby explore them.

Presently, the *Data Aspect* is commonly implemented in special databases, such as LMFDB, FindStat or OEIS, that use tables or lists of numerical of symbolic data. The *Software Aspect* is represented by mathematical computation systems, such as GAP, SageMath, and others that implement algorithms on top of this data. The *Knowledge Aspect* is implemented in mathematical documents that describe mathematical concepts, their meanings, and their properties. The latter aspect includes at least the knowledge underlying the data (e.g., the schemas of the database) and the software (e.g., the data types involved in the computations) and thus bridges between those two. An illustration with more examples of these aspects is given in Figure 1 (from the OpenDreamKit proposal).

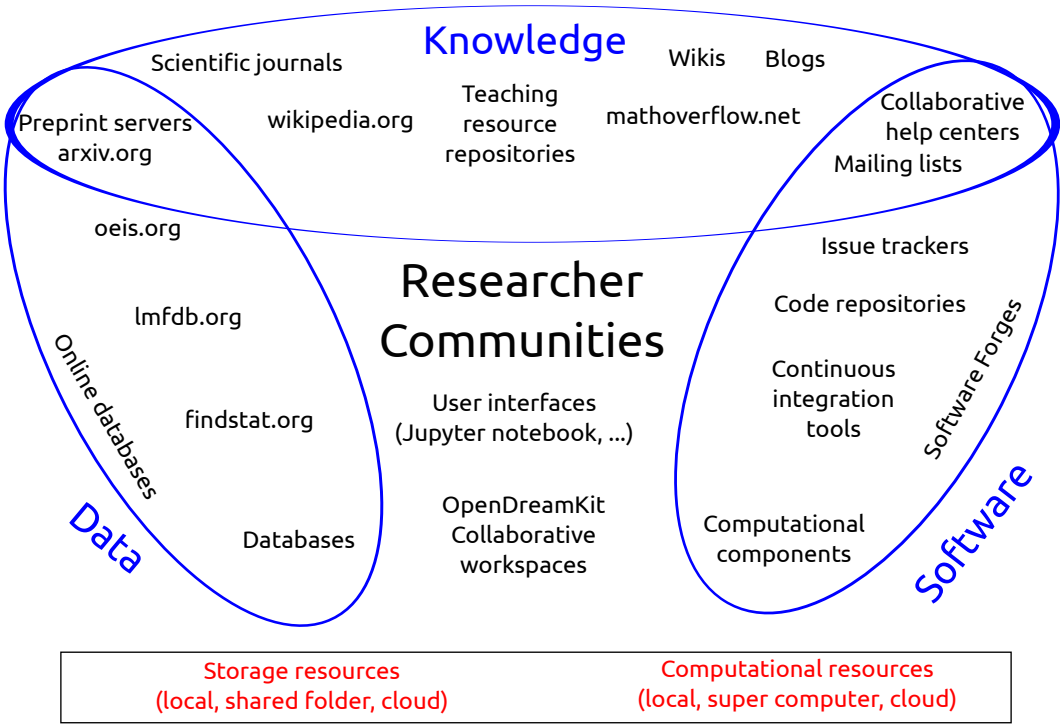

FIGURE 1. Virtual Research Environments for research in pure mathematics and applications.

Notably, even though most systems combine elements from more than one aspect, each system has a clear primary aspect and uses only optional or auxiliary infrastructure for the other aspects. For example, a computation system like SageMath includes knowledge mostly in comments and data mostly in dedicated tables. Similarly, a database system like LMFDB includes a simple knowledge base for the relevant definitions and performs some mathematical computations in its frontend. Thus, each system tends to be biased towards one aspect, and that makes interoperability between these systems unnecessarily difficult.

The approach of work package WP6: "Data/Knowledge/Software-Bases" in the OpenDreamKit project is to model these three aspects of mathematics research explicitly and uniformly. This provides a strong theoretical basis for the envisioned mathematical VRE toolkit. Concretely, the OpenDreamKit proposal calls for an extension of the well-established framework of theory graphs — developed for the representation of mathematical knowledge and languages (see Section 3) — with Data and Software components to arrive at a foundation for mathematical $\mathcal{DKS}$-bases. This deliverable report surveys the results of the first year and presents the initial design for the $\mathcal{DKS}$-bases.

In a series of workshops (September 2015 in Paris, January 2016 in St. Andrews, June 2016 in Bremen, and July 2016 in Białystok) the participants working on WP6 met and discussed the topic of integrating the OpenDreamKit systems into a mathematical VRE toolkit[1]. Key results were the observation that knowledge-aware interoperability of software and database-systems is critical for OpenDreamKit as well as the consensus that it can be achieved by aligning the mathematical knowledge underlying the various systems. This requires explicitly representing the three aspects of mathematical research and basing computational services and inter-system communication on a joint $\mathcal{DKS}$-base. These results are engrained in the "Math-in-the-Middle" (MitM) paradigm [**DehKohKon:iop16**], which is a central result of this report.

In the rest of this report, we describe the following:

(1) In Section 2 we report on the initial survey of existing OpenDreamKit-systems with $\mathcal{DKS}$ aspects (see also Appendix A for the raw data). This survey was conducted at the Paris workshop in September 2015 and through GitHub as part of the OpenDreamKit project. Moreover, we derive requirements for a $\mathcal{DKS}$ theory from that.

(2) In Section 3 we introduce the knowledge-driven OMDoc/MMT framework, which we use as the starting point towards building $\mathcal{DKS}$ theories. This is a previously existing knowledge representation language. Because it already abstracts from system and language-specific idiosyncrasies, it is well-suited for providing the level of abstraction necessary for a system-integrating VRE. Our goal will be to extend it to the data and software aspects.

(3) In Section 4, we present the MitM paradigm. This is a novel method that we have developed in OpenDreamKit for integrating the *software* systems into *knowledge* representation.

(4) In Section 5, we extend the MMT language and system with novel explicit concepts for representing the data aspect.

(5) In Section 6, we present a few case studies of representing concrete software and data systems on top of our $\mathcal{DKS}$-bases.

The resulting implementation of $\mathcal{DKS}$-bases on top of the MMT system provides the basis for general VRE services, which can be developed in future OpenDreamKit phases. These include remote procedure call via the SCSCP protocol [**SCSCP**; **FHKLR:SCSCP08**; **HorRoz:ossp09**], joint search engines over heterogeneous libraries of mathematical systems, or computational service discovery via MONET-like methods [**aird-et-al:2005**].

2. Report and Case-Study (for D6.2)

Starting in September 2015, we conducted a survey of the various big systems involved in OpenDreamKit. These would be SageMath and GAP for computer algebra(-focused) projects, and FindStat and LMFDB for database(-focused) projects. The OEIS is a well-known project

[1] It is notable that for many years many of the participants to these workshops had collaborated on a variety of projects, even occasionally on interoperability issues. For instance a previous workshop in Edinburgh in 2013 was titled *Online databases: from L-functions to combinatorics*, and grouped many participants to the SageMath, FindStat and LMFDB collaborations. However, the machinery of theory graphs was quite foreign to them then, and still needed to be extended to the Data and Software realms.

that is also focused on a database and was present in many discussions, but did not need to be explicitly surveyed. We nevertheless introduce that project in our summaries below. The raw survey results are in Appendix A

2.1. **OEIS**

OEIS [**oeis**] stands for On-Line Encyclopedia of Integer Sequences. It is a collection of around 250 thousand integer sequences that are stored as pure text form. The OEIS is licensed under Creative Commons and thus freely accessible.

2.2. **GAP**

GAP [**gap**] is a computer algebra system with a particular emphasis on group theory and discrete mathematics in general. Its fundamental ontology consists of *objects* (e.g. a monoid) satisfying various (composite or elementary) *filters* (e.g. *isAbelian*) which can be thought of as the *types* of objects. Filters can *imply* other filters. On top of these filters, operations are defined (e.g. computing the *degree* of a group) which are implemented by arbitrarily many concrete implementations called *methods*. The user only ever applies operations - the GAP system then uses a sophisticated method selection algorithm based on the specific (additional) filters satisfied by a given object.

GAP also has an extensive collection of associated databases.

2.3. **SageMath**

SageMath [**sagemath**] is a Python-based computer algebra system. Much of its knowledge is organized based on the notion of a *category* (which is related to, but not equivalent to the category theoretical notion), which provides *methods* on its *elements* (e.g. elements of a group), its *parents* (e.g. groups themselves) and its *morphisms* (e.g. the group homomorphisms). Each category can add new *axioms* and inherit from other categories - e.g. the category *AbelianGroups* inherits from *Groups* and adds the axiom *abelian*.

Much like the object/filter/operation tools in GAP, the category framework in SageMath is meant to provide uniform infrastructure to build mathematical objects. This is operationalised by dynamically constructing classes that represent parents, elements, morphisms, etc (and, crucially, their hierarchies).

Just as for GAP, a lot of the knowledge is embedded in the documentation. This seems to be due to the ease of access for developers, but also due to the need of documenting the implementation decisions made, which invariably are tied to very domain-specific knowledge. I

2.4. **LMFDB**

LMFDB [**lmfdb**] is a web service interacting with a database of objects from number theory. It is mostly focused on L-functions, but these can be obtained through many different constructions, so the LMFDB is also concerned with the many related objects. The database is built in MongoDB and as such uses JSON to model all of its data, and the web service interfaces to it mostly through custom Python or SageMath software for on-the-fly computations (or software packaged within SageMath), although there are efforts to switch to precomputed tables. A lot of the mathematical knowledge is implicit in that workflow.

The LMFDB includes a collaborative knowledge base, called *knowls*. Remarkably that knowledge base is important for the onboarding process to the collaboration.

2.5. **FindStat**

FINDSTAT [**findstat**] is a database of objects from combinatorics. These objects are of very few kinds: combinatorial collections, combinatorial statistics and combinatorial maps, each containing many collections. FINDSTAT is a clear extension to the idea underpinning the OEIS, and uses lots of SageMath code. Users of FINDSTAT are almost certainly users of the OEIS, and

tend to be SageMath contributors. FINDSTAT is also meant to be very collaborative, with a wiki attached to each of the collections.

2.6. **Observations**

The projects listed here are of two types: either data-backed or software-backed. In both cases, the mathematical knowledge is implicit, but communicated through different channels: software documentation for software projects, database schemas and wikis for data-backed projects. There is a lot of interdependency between those projects, both on the developer base and on the functionalities. SageMath, for instance, interfaces (or is interfaced to) by all the other projects.

3. THEORY GRAPHS: THE KNOWLEDGE ASPECT

We only sketch MMT here to the extent that we need it and refer to [**RabKoh:WSMSML13**] for details. An OMDoc/MMT theory graph is a diagram in the category of MMT theories and theory morphisms. Theories represent all kinds of languages such as mathematical foundations and type systems as well as individual mathematical theories. Theory morphisms represent relations between them including translations, imports, and representation theorems.

Theories. In the simplest case an MMT **theory** is a finite list of symbol declarations. Each **symbol declaration** must have a name and may optionally have a type object, a defining object, and arbitrary meta-data (such as tags, cross-references, comments, etc). The **objects** are OpenMath 2.0 objects [**BusCapCar:2oms04**] — these are complex expressions formed from application, binding, variables, literals, and symbol references.

Critically, MMT enforces structural validity: Every object may only reference symbols that are declared in or imported into the containing theory. But MMT abstracts from the foundational semantics such as type and logic systems that specify exactly which objects are meaningful. This puts MMT into a powerful intermediate position, where enough structure is guaranteed for knowledge management while not committing to a particular foundation.

An example MMT theory can be found in Listing 1. The theory `Int` shows a part of the formalisation of integers including comments. Here the namespace is just a URI that is used to form globally unique URIs for all subsequent knowledge items. As described above, each symbol declaration may consist of multiple components: types are prefixed with `:`, definitions with =, and notations with #.

LISTING 1. Example of an MMT theory: Defining a theory of integers.

```
/T MMT theories have an attached namespace
namespace http://www.opendreamkit.org/

/T We define a formalisation for integers
theory Int : ?LF =

  /T We make use of a logical foundation
  include ?Logic

  /T We declare a type of integers
  int: tp  # ℤ

  /T We define a minus operation
  minus : tm ℤ → tm ℤ  # − 1

  /T Now we define even more operations
  leq : tm ℤ → tm ℤ → prop  # 1 ≤ 2
  geq : tm ℤ → tm ℤ → prop
```

```
        = [a,b] b ≤ a  # 1 ≥ 2

/T And axioms on them
leq_refl : {a} ⊢ a ≤ a
leq_antisym : {a,b} ⊢ a ≤ b → ⊢ b ≤ a → ⊢ a ≐ b
leq_trans : {a,b,c} ⊢ a ≤ b → ⊢ b ≤ c → ⊢ a ≤ c

/T (omitted some more declarations here)
```

Above we have skipped the MMT module system, which is not essential for our results here. In the simplest case, include declarations import declarations from other theories. Above this is used to include the theory Logic into the theory Int. (The ? character occurs to form the URI of a theory relative to the current namespace.) Moreover, there is one detail of the MMT module system that is critical for OpenDreamKit: Every theory may have a **meta-theory**. Above, the meta-theory of Int is LF. Practically, the meta-theory mostly behaves like an include. But conceptually, the meta-theory of a theory T is the language that provides the foundational background to understand T. The most common use case employs three meta-levels: Firstly, an MMT theory introduces a logical framework F. Secondly, F serves as the meta-theory of a foundation L, which uses the symbols of F to define a particular type system and logic. Thirdly, a library of mathematical knowledge is developed as a theory graph in which all theories have meta-theory L.

A sample theory graph can be found in Figure 2. It contains some of the theories we make use of in our approach.

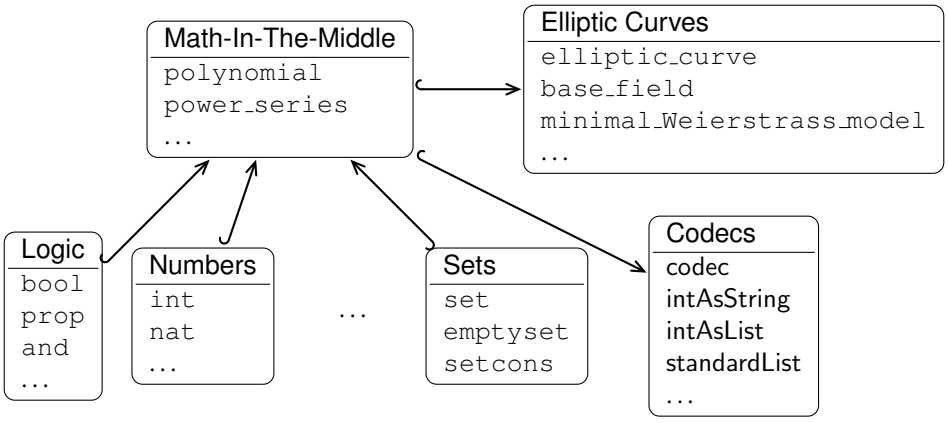

FIGURE 2. A Theory Graph of some of the theories (excluding their meta-theories) involved in OpenDreamKit. Symbols are listed underneath the theory name, and includes are represented as solid edges. We omit the full declarations and some of the more fine-grained structure for simplicity.

Most systems (including most systems involved in OpenDreamKit) focus on the third level only. The second level is usually left implicit, e.g. because it is hard-coded in the implementation of the computation system. Because the second level is hard-coded, there is then no need for the first level. For system integration, it is important to make the second level explicit so that the semantics of the exchanged knowledge can be specified. Thus, MMT already anticipates much of the high-level concepts needed for the system-spanning $\mathcal{DKS}$-bases in OpenDreamKit.

Theory Morphisms. There are two kinds of theory morphisms: imports and views. Imports are central for building theories modularly by pulling together symbols introduced in other theories. Practically, imports can be interpreted as copying symbol declarations while instantiating,

renaming, or otherwise modifying them. The semantics of theories with imports is defined categorially via colimits.

A view from theory S to theory T is a translation of S-objects to T-objects. Contrary to imports, views are given after the involved theories have been defined. Thus, they have the character of theorems rather than definitions, and indeed views usually induce proof obligations that must be discharged for a view to be accepted. In the presence of appropriate logical frameworks, MMT guarantees a critical property: the translation of objects preserves all properties of the type system and the logic. *In particular, all S-theorems are translated to T-theorems by views*.

Views are mainly used for the *integration of knowledge* that comes from different sources. Say system A defines a group as an associative quasigroup with identity over an operation $/$ (in a theory G_A), and system B as a monoid with inverses for an operation $\circ$ (in a theory G_B), then we can build a theory isomorphism (a pair of mutually inverse views) that relates the two group theories G_A and G_B and allows to transport all theorems between the two theories – and consequently all algorithms between the two systems A and B. The modularity of the OMDoc/MMT system and an elaborate calculus of theory morphisms make the establishment and management of views effective.

Implementation. Theory graphs are implemented in the MMT system [**Rabe:MAGMS13**; **uniformal:on**]. At its core, this system allows for the declaration of theories along with symbols, imports and views, to build objects and translate them along views.

On top of this, the MMT system provides a number of logical and knowledge management services. The former includes computing the respective colimits, translating objects along morphisms, or type checking and proving objects relative to the respective meta-theory. The latter include import/export of libraries, editing, browsing, and middleware for system integration.

4. MATH IN THE MIDDLE: THE SOFTWARE ASPECT (FOR D6.2)

The OpenDreamKit project intends to integrate multiple mathematical software systems into a VRE toolkit. These systems are constituted by large collections of algorithms manipulating highly optimized data structures representing mathematical objects with the intent of solving specific computational problems. These systems overlap in the mathematical objects they cover and the problems they can solve, but every system has aspects that are not covered by any other system (as efficiently or generally). In particular, algorithms, implementation languages, and data structures differ significantly between systems and are optimized to their particular domain and intended performance profile. As the systems represent decades worth of experience and development, a re-implementation is prohibitive in cost and might lead to systems with greater coverage, but less efficiency.

Given this situation, the integration problem in OpenDreamKit becomes a problem of establishing an interoperability layer between systems. As we have seen in the previous section, the mathematical knowledge — for specifying the computational problems — can be expressed and made interoperable via views in the OMDoc/MMT format, specifying the exact data structures and intended behavior of software systems — and possibly verifying that the implementation conforms to this is the realm of "Formal Methods". Again, the effort of doing this for any of the systems in OpenDreamKit is prohibitive and way beyond the scope of the project.

4.1. Specification of Interfaces

If we analyze mathematical software from a specification-based viewpoint, then we see three levels:

S1. Math Specification represents the underlying mathematical knowledge and the computational problems of the domain in a system-independent way.

S2. Interface Specification represents the interfaces of mathematical software systems: the abstract data structures, and the input/output behavior (and possible side-effects) of the user-visible functions and procedures provided by the system.

S3. Implementation gives concrete implementations of the interface specification in a specific programming language.

Most modern programming languages support the organization of programs into software libraries by separating the specification (**S2.**) and implementation levels (**S3.**), allowing multiple implementations of a single interface specification. Examples include abstract vs. concrete classes in object-oriented programming, signatures vs. structures in SML, header files vs. C files in C, and operations vs. methods in GAP.

In all of these languages, the interface specification level is utilized for intra-library interoperability, making use of the more abstract description of the interface specification that can be instantiated by its various implementations. The interface specifications usually tie the names of the interface functions to argument and result types.

The specification of intended behavior is usually left to documentation facilities. This is the domain of the math specification level (**S1.**). This level is only marginally supported by programming languages, but a central concern in the OpenDreamKit project. The math specification level differs from the other levels by using some kind of logical system that can express universal properties like $\forall x, y.x = \mathrm{sqrt}(y) \Leftrightarrow x^2 = y$.

Fortunately, we do not need to specify and verify all existing systems in order to integrate them into a VRE toolkit: the specification (of objects and intended behavior) at the mathematical and interface level is sufficient. In particular quality control (establishment of correctness of the implementations) can be left to other means[2] and as a result we can resort to more lightweight methods for establishing interoperability.

[2] In particular, it is independent of interoperability of mathematical software systems.

4.2. **The Math-in-the-Middle Paradigm**

In the OpenDreamKit project we want to cover the software aspect of a math VRE toolkit via an approach we call "Math-In-The-Middle" paradigm (MitM; see [**DehKohKon:iop16**] for details and Figure 3 for an overview diagram). In contrast to most programming languages, the MitM paradigm concentrates on levels **S1.** and **S2.**, represents them in the OMDoc/MMT format and leaves the implementation (**S3.**) to the respective systems.

Here the underlying mathematical knowledge (level **S1.**), the "real math", is used as a reference ontology (in the "middle" – hence the name) for the math VRE toolkit. This ontology is represented as an OMDoc/MMT theory graph M as described in the previous section.

Additionally, for every systen in the OpenDreamKit VRE toolkit we establish an interface specification as an OMDoc/MMT theory graph I and link it to the MitM ontology via **interface views**. These fulfil two purposes: they align the namespaces of the systems with the math specification and they specify the intended behavior of the systems in terms of the MitM ontology: recall that OMDoc/MMT views transport I-theorems to M-theorems, so all properties expressed in these are conserved.

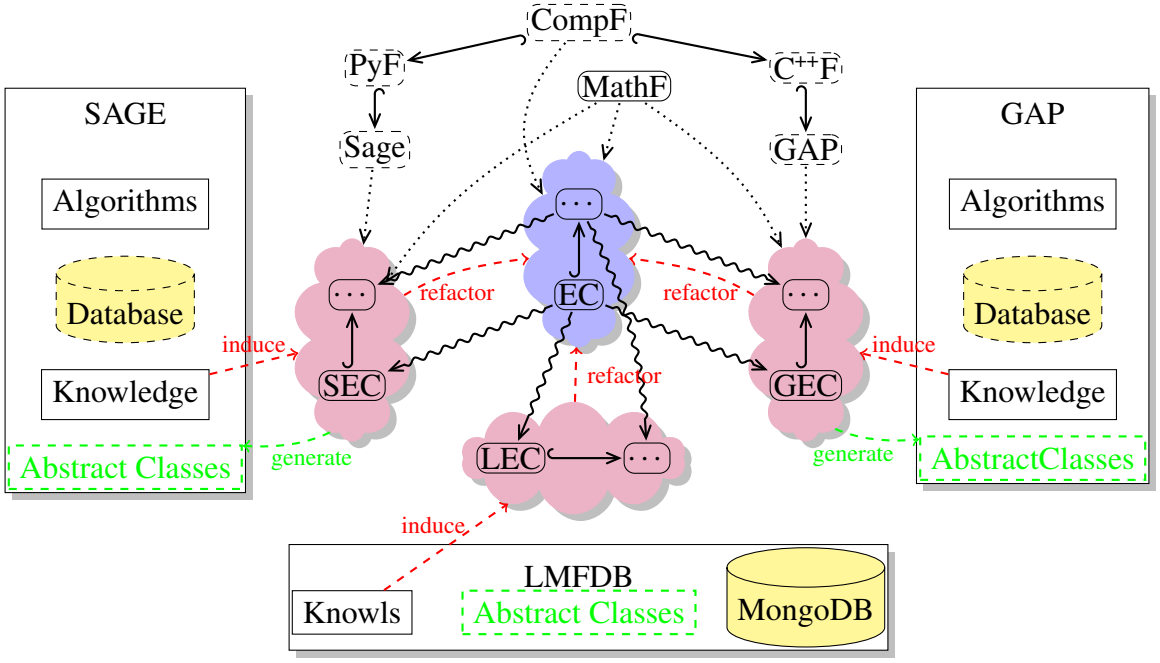

FIGURE 3. The MitM paradigm in detail. PyF, C^{++}F and CompF are (basic) foundational theories for Python, C^{++} and a generic computational model. SEC, LEC and GEC are theories for SageMath, LMFDB and GAP elliptic curves.

A sketch of the theory graph based on the example of elliptic curves can be found in Figure 2, an overview of the paradigm can be found in Figure 3. We will not go into details here but show how this architecture integrates the *Software* and *Knowledge Aspects*. Clearly, the MitM ontology – the blue cloud in the middle – is a specification of the underlying mathematical knowledge as an OMDoc/MMT theory graph, while the system interface theories – the purple clouds around it – formally specify the names and types (i.e. the argument patterns) and intended behaviour of the interface functions of the systems (often semi-formally to make the MitM approach scalable). The OMDoc/MMT views – the wavy arrows between the theories – are interpretation morphisms; in this particular case where they connect the mathematical specification to the system theories, they express the "implementation relation". Thus the OMDoc/MMT framework already allows to integrate the knowledge and software aspects for system interoperability.

4.3. Design Decisions and Initial Evaluation

The MitM paradigm we choose for the software (S) aspect of the OpenDreamKit VRE toolkit essentially takes two design decisions:

D1. Formalizing the interface specification (**S2.**: names and types of the interface functions) of the systems is sufficient to ensure system interoperability.

D2. Integrating the implementations at Level **S3.** — e.g. GAP or Python code — into the OMDoc/MMT theory graphs is overkill: this code can anyway only be executed by the respective systems — i.e. GAP or SageMath— and is rarely exchanged between systems.

Therefore we will base our foundation on OMDoc/MMT theory graphs directly rather than on an extension of OMDoc/MMT with "biform theories" [**KohManRab:aumftg13**; **Farmer:btc07**] as envisioned in the proposal. Such biform theories could explicitly represent the code as well and thus enable the (partial) verification of mathematical software systems, but this is not on the critical path towards a mathematical VRE.

The MitM paradigm constitutes a lightweight alternative; identifying and refining it has been one of the major achievements of the first year in WP6.

To evaluate the paradigm and the design decisions we have implemented extensions to the GAP and SageMath systems that export interface theory graphs in the OMDoc/MMT format (see Section 6 for details):

- GAP exports types, constructors, functions, data, and their documentation: 4097 Objects exported (2996 unique) in ca. 210 theories.
- SageMath exports categories/types, annotates functions: 382 Categories using 25 Axioms and (in total) 808 methods.

These interface theories allow the representation of all mathematical objects in GAP and SageMath as OpenMath2/MathML3 objects [**BusCapCar:2oms03**; **CarlisleEd:MathML3**] whose symbols are grounded in the interface theories (interpreted as OpenMath content dictionaries). GAP already had an **OpenMath phrasebook** — an import/export facility for OpenMath objects — and we have developed one for Python and SageMath [**py-openmath:on**].

Even though the development of the MitM paradigm is still at an early stage, these case studies show the potential of the approach. We hope that the nontrivial cost of curating an ontology of mathematical knowledge and interface views to the interface theories will be offset by its utility as a resource, and we are currently exploring this hope. The unification of the knowledge representation components in the MMT system

(1) enables VRE-wide domain-centered (rather than system-centered) documentation: the namespace alignment given by the interface-views allows to re-use documentation for a concept, object, or model in the MitM ontology in all interface functions aligned with it.

(2) can be leveraged for distributed computation via uniform protocols like the SCSCP [**HorRoz:ossp09**] and MONET-style service matching [**CaprottiEtAl:MathServiceMatching04:tr**] (the absence of content dictionaries — now given as interface theories — was the main hurdle that kept these from gaining more traction). Again, GAP already had an SCSCP interface, and we are developing one for SageMath at [**py-scscp:on**].

(3) will lead to the wider adoption of best practices in mathematical knowledge management in the systems involved; in fact, this is already happening: the development of the GAP interface theory exporter led to the discovery of hundreds of documentation errors and to a large-scale code refactoring that made type information more explicit and could lead to efficiency gains by extended static analysis in the future.

5. Virtual Theories: The Data Aspect

5.1. **Virtual Theories (for D6.2)**

OMDoc/MMT theories are limited when it comes to representing large amounts of data. Conceptually, every database should be represented as one theory, but this can easily lead to very large theories. For example, the theory for elliptic curves in the LMFDB would contain 319,882 symbol declarations: one definition for every curve. Prior to OpenDreamKit, MMT could only load whole theories into main memory, which made it insufficient as a basis for $\mathcal{DKS}$-bases.

Moreover, many data-driven theories are technically infinite collections of which only finite fragments have been explored so far. For example, this applies to almost all the databases in the LMFDB (each of which enumerates a certain infinite class of mathematical objects) and OEIS (each of which enumerates a certain integer sequence). As the explored fragments grow, the set of symbol declarations in the corresponding MMT theory must grow accordingly.

Therefore, we generalize MMT theories to allow for a virtual, possibly infinite set of declarations, that is explored dynamically. The combination of virtual theories with the Math-in-the-Middle approach yields our desired $\mathcal{DKS}$-bases.

Definition 1 (Virtual Theory)**.** A **virtual theory** is like an MMT theory but with a (possibly infinite) partially ordered set of declarations.

We give a trivial example of an infinite virtual theory for the natural numbers: besides the usual symbols for 0 and `succ` as well as the Peano axioms, it contains the totally ordered set of one declaration for every natural number. For example, we might have a declaration

$$5 : \texttt{nat} = \texttt{succ}(4)$$

to introduce a symbol for the number 5. In the presence of an addition operator, this theory might also contain one axiom for every pair of natural numbers, e.g., to state the truth of $2 + 2 = 4$.

This is a typical situation: we have an infinite (or very big) set of declarations that are generated systematically. In some cases (as for the natural numbers above), every declaration can be easily generated on-demand. Thus, one might think that virtual theories can be easily represented in a finitary way by storing the algorithm that produces the generations.

However, this falls short in general. For example, the generation of the declarations may be so expensive that it is only practical if they are precomputed and stored in a database. This is what happens in the LMFDB (and was why the LMFDB was introduced in the first place). It is also possible that there are multiple algorithms enumerating different fragments of the virtual theory, or that no generating algorithm is known (e.g., for some integer sequences in the OEIS) and individual declarations must be collected manually.

Listing 2. JSON representation of a curve in LMFDB.


```
{
    "torsion_structure": ["5"],
    "ainvs": ["0","-1","1","-10","-20"],
    "x-coordinates_of_integral_points": "[5,16]",
    "cm": 0,
    "number": 1,
    "rank": 0,
    "sha_primes": [],
    "galois_images": ["5Cs.1.1"],
    "heights": [],
    "torsion": 5,
    "iso_nlabel": 0,
    "aplist": [-2,-1,1,-2,1,4,-2,0,-1,0,7,3,-8,-6,8,-6,5,12,-7,-3,4,-10,-6,15,-7],
    "min_quad_twist": {"disc":1,"label":"11a1"},
    "sha_an": 1,
```

```
    "local_data": [{"ord_disc":5,"ord_cond":1,"kod":"\\( I_{5} \\)","ord_den_j":5,"p":11,"cp":5,"rootno":-1,"
        red":1}],
    "conductor": 11,
    "lmfdb_iso": "11.a",
    "2adic_label": "X1",
    "xainvs": "[0,-1,1,-10,-20]",
    "jinv": "-122023936/161051",
    "label": "11a1",
    "2adic_log_level": 0,
    "tamagawa_product": 5,
    "lmfdb_number": 2,
    "torsion_generators": ["(5, 5)"],
    "degree": 1,
    "2adic_gens": [],
    "torsion_primes": [5],
    "signD": -1,
    "real_period": 1.26920930427955,
    "isogeny_matrix": [[1,5,25],[5,1,5],[25,5,1]],
    "special_value": 0.253841860855911,
    "non-surjective_primes": [5],
    "lmfdb_label": "11.a2",
    "2adic_index": 1,
    "equation": "\\( y^2 + y = x^{3} - x^{2} - 10x - 20\\)",
    "gens": [],
    "regulator": 1,
    "sha": 1,
    "anlist": [0,1,-2,-1,2,1,2,-2,0,-2,-2,1,-2,4,4,-1,-4,-2,4,0,2],
    "iso": "11a"
}
```


For example, consider the database of elliptic curves in the LMFDB. A curve is typically defined by a JSON object such as the one in Listing 2. Not only does this record include a field label for a unique identifier and the defining equation, but it includes numerous additional values such as the conductor or the 2-adic generators, some of which can be very expensive to compute.

Implementation. In practice we never need to access all of the virtual declarations at once — in most scenarios we only need a very small subset of them, usually small enough to hold in memory. This motivates the main idea behind how we have implemented virtual theories in the MMT system.

MMT already abstracts from physical storage backends (working copies, databases, etc.), from which theories are loaded. We have extended this storage abstraction to allow loading not only theories but individual declarations on demand. This is more difficult than it sounds because while theories have a self-contained semantics, declarations only make sense in the context of the containing theory. Thus, we had to comb through the MMT code base and generalize all processing to the case where a theory's declarations are only partially known.

We have also built an LMFDB-specific implementation of this this generalized storage abstraction. This instance dynamically queries LMFDB for the appropriate entry, computes the corresponding declarations, and adds it to the in-memory representation of the virtual theory. (Additionally, if that virtual theory is not in memory yet, it first creates it.) We were able to retain an important feature of MMT: the loading of declarations is transparent to the user, and the storage loads a declaration automatically when and if it is needed by some operation.

5.2. Relating Database Objects and Mathematical Objects (for D6.3)

The storages sketched in Section 5.1 are not as simple as one might think. A major complication is that scalable databases only provide relatively low-level data types. For example,

a typical relational database provides primitive types for, e.g., integers and strings, and tables contain records built from these. JSON databases (as used in MongoDB [**Chodorow:mdg10**], which is used in LMFDB) or XML databases are slightly better by providing structured types like trees and lists. But the sets of mathematical objects stored in mathematical data systems use much richer data types such as matrices, polynomials, permutations, and arbitrarily more complex types built from them.

Therefore, any data system must employ encodings that translate the actual mathematical objects into database objects. This has been done ad hoc in the past and has proved both very difficult and — due to differing or undocumented encodings — an obstacle for system interoperability. Therefore, we have developed a systematic method for encoding/decoding mathematical objects as database objects. This allows formally specifying the schema of a database in such a way that MMT storages can use it to encapsulate the encoding and provide users with a high-level view of a mathematical database. A sketch of our method can be found in Figure 4 — we will give a detailed explanation below.

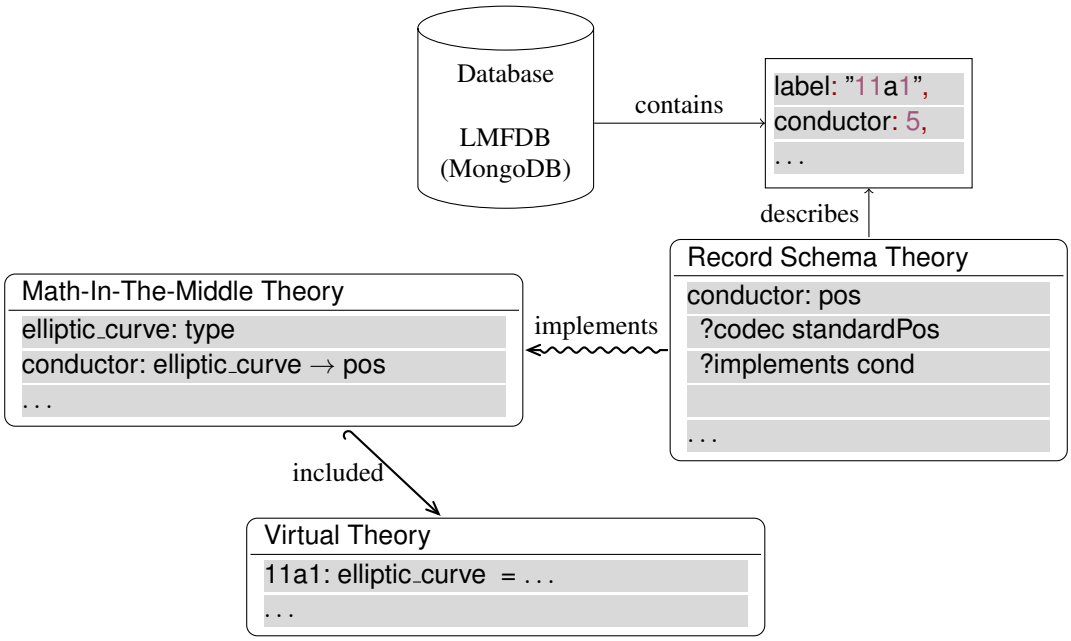

FIGURE 4. Translation between Database and Mathematical Objects: The example shows the elliptic curves database of LMFDB

5.2.1. *Codecs.* We fix an arbitrary data type of **codes**. These are the primitive values of the database. Typical examples are strings or JSON objects. We will use JSON for the purpose of giving concrete examples.

Our goal is to define codecs that define the relation between MMT objects and codes.

Definition 2 (Codec)**.** For an MMT object t, a t**-codec** is a pair $(\texttt{encode}, \texttt{decode})$ where

- `encode` is a partial function from MMT objects of type t to codes
- `decode` is a partial function from codes to MMT objects of type t

such that `decode` and `encode` are inverse to each other whenever defined.

Both encoding and decoding are partial functions. This is to be expected for decoding: only certain codes are the result of encoding objects of type t. It may be surprising for encoding. Here we need partiality because MMT objects may be arbitrarily complex expressions, not all of which denote mathematical values and can be encoded easily. This can include non-normalized expressions or expressions with free variables or uninterpreted symbols. We will see some examples below.

Recall the Math-in-the-Middle theory from Section 3. A trivial example of an int-codec is essentially the identity function: `encode` maps integer literals to the corresponding JSON integer, and `decode` inverts it. `encode` is partial because it only encodes literals, e.g., it does not encode expressions like $x + 1$ or $\min\{x^4 + x^3 + x^2 + x + 1 | x \in \mathbb{N}\}$. (An object like $1 + 1$ can be encoded by first simplifying it to a literal 2.)

One might think that it is sufficient to fix one t-codec for every type t. However, that is not realistic. In fact, already for the seemingly-trivial case of integers, we need multiple different codecs.

First of all, note that the int-codec given above cannot encode integer literals that do not fit into the 64 bit integers of a typical JSON implementation. In LMFDB databases, we indeed encounter such integers very often. Therefore, we have specified two additional codecs. intAsString encodes integer literal as strings in decimal representation. intAsString has the advantage of easily encoding all integers, but it is not convenient for computations. Therefore, LMFDB users have occasionally used a smarter encoding: intAsList encodes integer literals as lists of JSON integers such that the list $[n, d_1, \ldots, d_n]$ corresponds to the integer whose base 2^{64} representation is $(d_1 \ldots d_n)$. intAsList has the advantage that the lexicographic ordering can be used for size comparisons without having to decode (this requires first comparing the length of the sequences, encoded).

Along the same lines, we can define codecs for other simple data types. For example, we can define a codec for matrices of integers that encodes pairs of pairs of integers as JSON lists of 4 JSON integers. This example is interesting because it comes up in the LMFDB and has previously caused confusion: because the encoding was not fully documented, users mistakenly assumed that the mathematical type of the field is list of integers rather than pair of pair of integers.

LISTING 3. Fragment of the special Codec Theory in MMT

```
namespace http://www.opendreamkit.org/

theory Codecs : ?LF

  /T We include the Math theory to be able to use types
  include odk:?Math

  /T Declare a type of codecs
  /T There can be multiple codecs for each type
  codec : type → type

  /T a codec that codes strings as JSON strings
  standardString: codec string

  /T a codec that codes integers as JSON integers if
  /T within 32 bits, and as JSON strings otherwise
  standardInt: codec int

  /T a codec that codes natural numbers (with 0)
  /T as JSON integers if within 32 bits, and as
  /T JSON strings otherwise
  standardNat: codec nat0

  /T a codec that codes positive natural numbers
```

```
/T as JSON integers if within 32 bits, and as JSON
/T strings otherwise
standardPos: codec pos

/T a codec that codes integers as a JSON array
/T of integers each of which fits within 32 bits
intAsArray: codec int

/T a codec that codes booleans as JSON
/T booleans
standardBool: codec bool

/T a codec that codes booleans as JSON integers
/T 0/1
boolAsInt : codec bool

/T a codec operator that codes lists as JSON
/T arrays, recursively coding each element of
/T the list
standardList: {A} codec A → codec (list A)

/T a codec operator that codes vectors of
/T fixed length as JSON arrays
standardVector: {A,n} codec A → codec (vector A n)

/T a codec operator that codes matrices of
/T fixed size as JSON lists of lists
standardMatrix: {A,n,m} codec A → codec (matrix A n m)
```

We collect and document all available codecs in a special MMT theory, a fraction of which can be seen in Listing 3. Here, codec t is the type of codecs that can encode expressions of type t.

Note that our codec theory does not include any implementations (which usually require highly programming language–specific function calls. Instead, it standardizes names for the codecs and documents their behavior so that each codec can be implemented faithfully in multiple programming languages. This is in line with the Math-in-the-Middle approach of centrally storing the shared knowledge.

We have seeded this theory with a few important codecs. And for every codec we describe, we have already given a reference implementation in Scala [**scala:webpage**] so that the MMT system (which is written in Scala) can use all codecs. In the future, OpenDreamKit will gradually build a library of relevant codecs and implement them in multiple programming languages.

Codec Operators. int is an atomic type, but mathematical types are usually very complex types. Already, types like List(int) provide substantial encoding problems because both integers and lists can be encoded in multiple ways. To systematize these choices, we introduce codec operators.

Definition 3 (Codec Operator)**.** For an MMT symbol t and an arity n, a t-codec operator takes an o_1-codec C_1, ..., an o_n-codec C_n and returns a $t(o_1, \ldots, o_n)$-codec.

For example, Listing 3 declares a symbol List, which maps t-codecs to List(int)-codecs. We specify it as the following codec operator standardList of arity 1. It takes a type t-codec C and returns the following codec: `encode` maps the object $\mathsf{Cons}(a_1, \mathsf{Cons}(a_2, \ldots, \mathsf{Cons}(a_n, \mathsf{Nil})\ldots))$ to the JSON list $[e_1, \ldots, e_n]$ where each e_i is the result or encoding a_i with C. `decode` is defined accordingly.

Similar to the codecs, the special MMT codec theory (Listing 3) also includes standardised names for these. Apart from standardList, we also have straightforward codec operators for vectors (as lists of fixed length) and matrices (as lists of lists of fixed lengths).

5.2.2. *Schema Theories.* Codecs can transform between MMT objects and codes, but we still have to specify which codecs to use for which types. We use special theories for this purpose, which we call *schema theories*. These satisfy two functions.

Firstly, a schema theory describes the database schema: many databases (including the ones in LMFDB) can be seen as sets of records conforming to a certain schema. We represent these schemas as MMT theories with one symbol declaration for each field. The meta-theory of these theories is the math-in-the-middle theory so that the types of the symbols can be the intended mathematical types. Secondly, we annotate meta-data to each declaration, providing the information which concept in the math-in-the-middel theory a field implements and which codecs to use for converting between the two.

Thus, the schema documents both the mathematical meaning of the fields and the physical encoding used when exchanging records conforming to the schema. By giving the schema theory for each database, we can capture all knowledge necessary to automatically interface with it.

LISTING 4. Fragment of the Elliptic Curve Schema Theory in MMT

```
namespace http://www.lmfdb.org/schema/elliptic_curves

/T This file documents the schema of the elliptic_curves collection of the LMFDB (see
/T http://lmfdb.org) in terms of an MMT theory. The documentation (and source) is in
/T https://github.com/LMFDB/lmfdb-inventory/blob/master/db-elliptic_curves.md

theory curves : odk:elliptic_curves?elliptic_curve =
  include odk:?Codecs

  meta /?Metadata?implements elliptic_curve?elliptic_curve
  meta /?Metadata?constructor elliptic_curve?from_record
  meta /?Metadata?key "label"

  degree : pos
    meta /?Metadata?codec standardPos
    link /?Metadata?implements odk:elliptic_curves?modular_degree?
      modular_degree

  2adic_gens: list (matrix int 2 2)
    meta /?Metadata?codec standardList (standardMatrix standardInt 2 2)
    link /?Metadata?implements odk:elliptic_curves?Rouse_classification?
      rouse_generators

  /T (omitted some more declarations here)
```

As an example, the schema theory for the database of *elliptic curves* in LMFDB is indicated in Figure 4. A larger fragment is given in Listing 4. It has meta-data linking it to the *type* of elliptic curves in the corresponding math-in-the-middle theory, as well as meta-data telling MMT which field in the database to use as *names* for the resulting MMT declarations — in this case the field `label`, which corresponds to a unique LMFDB-internal naming scheme. The schema theory contains e.g. a declaration `degree` of type `pos` (for positive integers), corresponding to a field in the database by the same name, which is annotated with meta-data telling MMT

(1) to use the codec `standardPos` to convert from (and to) an LMFDB entry, and
(2) that the field `degree` implements the function `modular_degree` in the math-in-the-middle theory.

We have implemented a new component of the MMT system that takes an expression c of type codec t and builds the appropriate t-codec by traversing c. We use this in the storage instance for LMFDB as follows:

(1) A declaration with name n in theory T is requested (e.g. the elliptic curve with label `11a1` in the virtual theory `lmfdb:db/elliptic_curves?curves`).
(2) We load the schema theory S for T (e.g. `lmfdb:schema/elliptic_curves?curves`)
(3) We connect to LMFDB and retrieve the corresponding record (e.g. the database entry for the curve with label `11a1`).
(4) We decode every field of the record according to the codec specified in S.
(5) We collect the decoded MMT object in an MMT record r, the mathematical representation of the requested object.
(6) We add the declaration $n : \mathsf{elliptic\_curve} = r$ to the corresponding virtual theory.

5.3. Towards Mathematical Querying of Databases (for D6.3)

So far we have only concerned ourselves with accessing virtual theories one declaration at a time. Going beyond the scope of the present deliverable, it is desirable to also implement a *querying* operation for virtual theories. Conceptually, this means to ask the MMT backend to load all those declarations of a virtual theory that satisfy a certain property. This operation is much more difficult because the naive approach of loading all declarations and then filtering them does not scale at all.

In the future of the OpenDreamKit project we want to investigate this question further. At this point, a result of OpenDreamKit is to sketch and discuss possible solutions to this problem.

Encoding Queries. A straightforward solution is to translate the property into a query expression that the underlying database can evaluate efficiently. This is similar to encoding objects as presented in Section 5.2 except that we have to encode complex expressions instead of only atomic values. Our current implementation already anticipates this solution, but we have not implemented the specific connection to LMFDB yet.

An example for a practically relevant query that is covered by this approach is to retrieve all elliptic curves whose conductor is 5. However, this approach is limited by the capabilities of the underlying database. For example, MongoDB does not allow efficiently retrieving all elliptic curves whose conductor is divisible by 5.[3]

In other data systems, there may not even be a strong query language. For example, OEIS stores sequences in semi-normalized text files. Therefore, it can answer a fixed set of query operators that combine text search with integer sequence–specific search in the precomputed prefixes of the sequences. This allows querying for all OEIS sequences whose precomputed prefixes contain the integers 5 and 10 with any sequence of numbers in between. But it cannot retrieve, e.g., all sequences containing n, $n+1$, $n+2$ for some n.

[3]This particular query was posed to John Cremona, who curates the elliptic curves database, by a mathematician. He was unable to carry out the query directly in the current architecture.

Indexing Sets of Values. Going beyond the above approach, we can supplement existing database with custom mathematics-specific indices. For example, we can easily create an index of all integers that occur anywhere in a database. This would have the additional advantage of allowing queries across different databases, e.g., to find occurrences of the number 142857 anywhere in LMFDB, OEIS, or any other data system.

The key design question here is what index to use. Indexing all integers is the simplest possible attempt.

For every occurrence of an integer, the index can store additional information that can be used in the query, e.g.,

- which database does it occur in,
- which function does it serve (conductor, sequence element, etc.),
- what values occur nearby.

An orthogonal improvement is to add for each integer in the index its factorization to the index. This would allow fast queries based on divisibility. Naturally, this yields a trade-off problem between the complexity of the index and the complexity of the queries that can be answered. This problem is well-known in databases in general but has not been studied for mathematical data.

The most difficult design question is how to extend such an index to more complex values such as complex numbers, sequences, or polynomials. Indeed, finding occurrences of polynomials, or simply storing an index of known factorizations of big polynomials is a concrete service currently needed by mathematicians.

Here the design space includes a gradual transition from indexing values to indexing mathematical expressions. An integer-only index can be seen as the extreme case of the former. The extreme case of the latter — indexing arbitrary expressions — has been explored already in the MathWebSearch system D6.1; it is optimized for substitution and unification queries, which are different from the integer-oriented queries discussed above. We conjecture that after realizing an integer-only index, a key question will be how to combine the best of both kinds of indices.

Composite Queries. The above has discussed only atomic queries. Composite queries arise when we use operators such as in SPARQL, XQuery, or SQL to join, intersect, filter, or translate query results.

Here two cases must be distinguished. Firstly, it may be possible to factor the query into atomic queries whose results can be aggregated into the overall result. For example, we might ask for all elliptic curves whose conductor is divisible by 5 and whose defining equation unifies with a certain pattern. Such a query can be evaluated by taking the intersection of the two atomic queries. This can be handled comparably simply. For example, the query language developed for MMT in [**Rabe:qlfml12**] already combines ideas from XQuery and SPARQL to compose queries from atomic ones.

Second, it is much harder to answer queries that do not factor. For example, we might ask for all elliptic curves whose conductor occurs in a certain sequence in the OEIS. Barring an unrealistically sophisticated index, such a query would require iterating over all elliptic curves or all entries of that sequence, and evaluate another query for each one. Either one of these iterations might be prohibitively expensive.

6. Case Studies (for D6.3)

While our theoretical model of $\mathcal{DKS}$-bases theories and our architectural design of virtual theories are applicable to a wide variety of systems, in the scope of OpenDreamKit we want to conduct a few case studies and connect to some databases in particular. These case studies are not formally part of this deliverable (they are mostly in D6.4), but we give a short overview of the current state below, except for FindStat, where no progress has been made so far beyond the survey.

6.1. GAP

The interface specification (Level **S2** from Section 4) for GAP consists of two parts. Firstly, we have a manually-written theory for GAP's ontology in MMT containing declarations for all of the above concepts. This serves as a meta-theory for a large number of theories that automatically generated from the GAP library. The result is currently a set of 4097 filters and operations as MMT symbols collected in approximately 200 theories. So far, all the imported operations have no information about their return types (i.e. the filters that apply to the returned object) because those are not specified by GAP. Currently, work is ongoing on GAP to make that information available within GAP in general and thus for the interface specification in particular.

As a first knowledge management application of this representation, we used MMT's generic graph display components to display the implications between GAP filters.

6.2. SageMath

For the interface specification, we proceed in the same way as for GAP. The manually-written meta-theory defines all the above concepts, and we automatically generate theories from the SageMath categories. The latter yields 274 categories with 25 axioms and 569 methods, where each category corresponds to one MMT theory declaring its methods and axioms as well as the corresponding documentation. The theory graph of the resulting theories mirrors exactly the inheritance graph of the original categories in Sage.

6.3. LMFDB

We have already implemented the schema and codec architecture above to build a virtual theory for the databases in LMFDB and — and as a guiding examples — written a first schema theory for the database of elliptic curves.

In the future we want to extend the coverage of the approach. This will include writing more schema theories and possibly introducing more codecs. This will likely also lead to some refactoring inside LMFDB itself, as the community for the first time will try to semantically describe its entire dataset.

6.4. OEIS

We have already semantified the pure text format and, among other things, this has helped us finding new relations between the existing sequences. A more detailed look at our previous work can be found in [**LuzKoh:fsarfo16**] and we will not go into details here. So far these efforts have helped us to understand how the OEIS database is structured.

Similar to LMFDB we plan on integrating this into our virtual theories architecture. We are considering building one DK theory per sequence, where the declarations in each theory contain the known elements of the sequence. We also plan to integrate this with our infrastructure on knowledge management services, such as MathHub and MathWebSearch.

7. Outlook and Conclusion

In this report we have outlined the initial design of a $\mathcal{DKS}$-base (the main objective of WP6: "Data/Knowledge/Software-Bases"). The basis is the OMDoc/MMT format that allows a foundation-independent representation of mathematical knowledge in theories that are linked by imports for modular development and views for integrating knowledge from different sources by interpretation. As anticipated in the OpenDreamKit proposal, the OMDoc/MMT format and its implementation in the MMT system is sufficient to for the knowledge ($\mathcal{K}$) aspect of $\mathcal{DKS}$-bases.

The OpenDreamKit workshops and the survey (see Section 2 and Appendix A) clarified that the system integration task is much more critical in the development of a VRE toolkit than the classical "Formal Methods" tasks of software verification or synthesis. As a consequence we base our integration of the software aspect ($\mathcal{S}$) on the specifically-developed "Math-in-the-Middle" paradigm, which constitutes a much more lightweight approach by concentrating on abstract (mathematics-level) specifications. We were able to show that this can represent this OMDoc/MMT theory graphs, if we can generate interface theories for the OpenDreamKit systems and align them to a mathematical reference ontology (the MitM ontology) via OMDoc/MMT views. We have established the feasibility of this by generating interface ontologies for the GAP and SageMath systems and conjecture that carries over to other OpenDreamKit systems.

For the data aspect ($\mathcal{D}$) of $\mathcal{DKS}$-bases we had to extend OMDoc/MMT to allow *virtual theories* that consist of arbitrarily many declarations and the MMT system to handle them efficiently, – loading only small subsets into system memory for processing. Thus we can connect to external databases which we model as a set of well-typed records, that is list of (key, value) pairs. We introduced the concept of record types inside MMT, keys are symbols which are declared inside a schema theory and values are MMT literals translated from the physical database representations using codecs. We used the LMFDB database of elliptic curves as a case study to test this approach and implemented a multitude of codecs as a "database connector" that lifts the database contents to virtual declarations in virtual theories that can be seamlessly integrated into the MMT system.

In the future we want to expand on both the implementation and the concept as a whole. Right now we can only translate database records into MMT objects. While we want to use the form of the objects used by MMT as the primary representation we want to be able to translate these objects to system specific objects, thereby building true *DKS* theories. Each system might have system-specific constructors and / or representations. In practice all systems will have a constructor for these objects. These will take a set of arguments. These arguments will either be primitive (in which case we can just encode them from MMT using a codec) or be complex objects themselves (in which case we can recurse into the entire procedure). Storing these encodings inside MMT we will be able to write thin interfaces to MMT, which can then easily retrieve objects from MMT in their preferred representation.

Together with the opposite process – the understanding of objects by using accessors from arbitrary systems – will also allow (almost) arbitrary systems and databases to exchange objects via MMT. We are already working on a Python Client implementation. This will not apply any recoding to the objects – it just retrieves records in an easily accessible form from MMT. In the future we are hoping to use this to integrate MMT and GAP and enable GAP to use any kind of object that MMT has access to.

It is interesting to note that both the integration of the $\mathcal{S}$ and $\mathcal{D}$ aspects into OMDoc/MMT necessitate the generation of theories: pre-generated as interface theories in the MitM paradigm, and on-the-fly in virtual theories.

APPENDIX A. RAW CASE STUDY RESULTS (FOR D6.2)

The authors are grateful to John Cremona, Alex Konovalov, Markus Pfeiffer, Viviane Pons, and Nicolas M. Thiéry for their time in answering our survey. Their responses and the structure of this survey have been adapted to fit this presentation format.

A.1. FINDSTAT

A.1.1. *Overview at a high level of the system*

FINDSTAT [**findstat**] is a database and a web interface accessing the database. It is designed by and for combinatorists. The purpose is to store and search information on *statistics* over *combinatorial objects*. A statistic is mostly a map between a set of combinatorial objects to the natural numbers. As an example, the number of edges is a statistic on graphs. The main purpose of FINDSTAT is to give an interface for one to *search* for some statistics the same way one would search for integer sequences on the OEIS [**oeis**].

A.1.2. *Available data*

A.1.2.1. Structure of the data

FINDSTAT has basically 3 categories of objects.

The combinatorial collections: FINDSTAT stores a list of combinatorial collections: 18 as of today (January 2016). All these combinatorial collections are actually linked to a SageMath combinatorial collection. We only store the minimal information needed to print the collection on the website and to recreate the collection in SageMath.

For every collection, we store a list of combinatorial objects. More precisely, we use SageMath to generate the list of objects, but we store a standardised version of the printout of the object. This standardised version is homemade: it has to be

standardized: a single given graph will always be printed the same way,

unique: two different graphs will never be printed the same way,

human readable: when possible, it should be easy to understand for a human reader and not only a machine (so no hash-key or anything like this). When possible, we keep the default printout of SageMath object. Sometimes, we have to store a little bit of code to convert this printout into a SageMath entry.

The combinatorial statistics: A statistic is a list of couples : combinatorial object from a certain collection or value. As of now, we have 364 statistics, each of them containing between 200 and 1000 entries. For each statistic, we store some metadata: name, identifier (specific to FINDSTAT, can be referenced from outside), combinatorial collection, description, code, references, etc. And we also store the data itself: a list of entries, each entry is made from combinatorial object (as a string, by its standardised printout) and a integer value. As an example, the values of "The number of edges of a graph" St000081 is a list of all graphs up to size 6 with their associated number of edges.

The combinatorial maps: A combinatorial map is a mathematical function from a combinatorial collection to another combinatorial collection. For example: binary search tree insertion turns permutations into binary trees. As of now, we store 107 maps each of them containing between 200 and 1000 entries. We store the metadata of the map: domain, codomain, description, code, etc. And we store the map-data as a list of (value, image) where value and image are combinatorial objects stored as strings through their standardised printout.

As an addition, FINDSTAT also provides some wiki pages with information on combinatorial objects, maps and statistics in a less formalized way.

The low level data format is a SQL database where we store everything we need. Most of the data described above is accessible through the website in HTML. All information about combinatorial statistics and combinatorial maps can be accessed through JSON files that have

standardised URLs depending on the identifier of said statistics or maps. It is possible that the URL changes if the website organisation is changed in the future but it will always be related to the identifiers which are set once and for all. The format of the JSON files are also likely to change but we try to limit those changes and keep backward compatibility as much as possible. Those JSON files are the closest we have to an external API, they are used by the SageMath-FINDSTAT interface.

All our data are distributed under Creative Commons Attribution-ShareAlike 3.0 Unported License.

A.1.2.2. How is this data produced? How is it changed?

The data are produced and changed through user contributions. As for now, 55 people are listed as contributors. We have an HTML form to submit statistics where the user receives many information on what should be submitted and in what format. Once a new statistic is submitted or a change is proposed, it has to be validated by one of the day developers. We don't receive that much data so the process is usually very quick. Each change is stored and so we have access to the full history of the statistic information with authors.

For maps, we don't have yet the "Add Map" form. Each map has to be added by one of the FINDSTAT developers. The reason is just that the maps are a more recent addition and so the adding process has not been finalized yet.

A.1.2.3. How do you document it?

We have a very basic documentation for statistic data that we provide to the user who which to contribute. We don't have any documentation for our dataformat (JSON files).

A.1.3. *What knowledge do you have in the system?*

A.1.3.1. What are the sources of external knowledge?

We rely on the knowledge of our contributors about statistics and maps and try to store it. We also depend on some SageMath algorithms, for example to generate the combinatorial objects.

A.1.3.2. Can you point to implicit knowledge? Is it common knowledge?

Our website is targeted at combinatorists. Even though we try to give all the basic definitions and information, it might be difficult to use for someone who has no knowledge of these objects.

A.1.3.3. What would you gain if it was made explicit/machine actionable?

At the moment, our infrastructure is really SageMath oriented (object printouts, names, etc). A language-neutral description of our objects might make it easier for interfaces from other system to appear. The gain for us is that the more user we have (from different background), the more contributors we might get and so the more mathematically pertinent our database is.

A.1.3.4. Have you gone in this direction? How did you represent the knowledge then?

Giving access to the statistics and maps data as JSON files was a first step in this direction.

A.1.3.5. How do you collaborate on knowledge representation?

By referencing those data (statistics and maps) and proposing unique identifiers that can be referenced from the outside (the same way the OEIS identifies integer sequences with a unique number).

A.1.4. *What software do you have?*

A.1.4.1. What custom software are you running?

We need the software SageMath to run some computations: basically, generating the objects, printing them, etc. The statistic and maps code are usually integrated into SageMath for consistency but it is not mandatory.

There is also some FINDSTAT specific code to run the website. Most of this code is just basic web-programming views of our database.

The database search is the heart of the service. It is a small algorithm that takes a user-given statistics and compares it to the database up to some maps.

A.1.4.2. In which language is your system written?

Our server runs on SageMath with some imported web packages, so it is written in Python. We use the Python wiki server MoinMoin as a backend and have written some customized MoinMoin plugins to run our service.

A.1.4.3. How does it use the data and the knowledge?

The data is stored in a SQL database. It is preloaded and precomputed when we launch the server then all computations are made on this preloaded data. We don't use the knowledge at this stage, we just basically request the database and compare numbers using some parameters. In the future, we might want to use the knowledge we have on the maps (bijection, injection, surjection, involution, etc) to improve our algorithm.

A.1.4.4. How does your software act on represented knowledge?

The software might put into light some mathematical relations between combinatorial objects but doesn't store them or anything like this.

A.2. **SageMath**

A.2.1. *Overview at a high level of the system*

SageMath [**sagemath**] is general purpose computational (pure) mathematics software. It has 300 contributors and consists of 1.5 million lines of Python/Cython code, around 40000 function, and 4000 classes. It is distributed with hundreds of open source (math) software and libraries.

Most of the survey answers are given specifically for the SageMath category framework, which is used to structure a lot of SageMath code by exploiting as much as possible of the underlying mathematical structure.

A.2.2. *Available data*

SageMath interfaces with a large collection of (optional) databases, usually coming from external software and possibly repackaged from external databases. Examples include GAP databases, the OEIS [**oeis**], various databases of elliptic curves, etc.

The data format is heavily reliant on pickling (Python protocol for serialization). Objects can be converted to strings and reconstructed. This is used for persistence, for storing in SageMath databases, for data exchange between SageMath instances. SageMath comes with code to reconstruct the object and perform sanity checks. By default, pickling is done by class and stores plain data (no encapsulation). We aim for pickling by construction (which would require more semantics).

A.2.3. *What knowledge do you have in the system?*

The system effectively knows many mathematical properties and theorems, algorithms, ...

In designing the system, a few key points conditioned the design:

(1) There are only a handful of fundamental concepts: operations (*, +, ...), axioms (associativity, commutativity, ...), ...;
(2) The richness arises in the combinations of these concepts (*e.g.* fields);
(3) We use an existing language and its object oriented features for modelling and method selection.

A.2.3.1. Sources of external knowledge?

Each SageMath contributor brings on specific mathematical knowledge about the objects studied, which might not be available to others in the collaboration.

A.2.3.2. Can you point to implicit knowledge?

The algorithms rely heavily on the mathematical properties of the objects they manipulate. SageMath uses the Object Oriented features of Python. The properties of a SageMath object are specified by its Python class:

- what mathematical object does it represent?
- how is it represented?
- the class information is often of technical flavor, and complemented by additional information on its universe (parent, category)

SageMath strives to model mathematics closely: not only matrices are instances of a specific classes and not plain list of lists, but linear maps themselves are instances of specific classes and not just represented by matrices. This reduces the risk of calling a meaningless function.

The abstract algebraic properties of an object (*e.g.* being a group or a field) are modelled relatively explicitly: objects know the names of their categories and axioms. The meaning is essentially implicit except, in the good cases, informally in the documentation and as testing methods. The names of the operations are hardcoded, which leads to duplication (for instance between additive and multiplicative structures). The size of the code is linear in the number of methods, which in the current setup ought to grow exponentially as the complexity of the modeled objects increases.

It is not always made explicit which methods an object in a given category should implement: methods and operations are documented, but their exact specifications is not always completely defined or defined consistently across the class hierarchy.

Some theorems (*e.g.* Wedderburn) are embedded in actionable form, but that information cannot be extracted or operated on.

A.2.3.3. Is it common knowledge?

The meaning of the relevant categories and axioms (*e.g.* ring or associativity) is relatively well known by the users and developers.

A.2.3.4. What would you gain if it was made explicit/machine actionable?

- Dynamic generation of documentation that the user can navigate
- Sanity/correctness checks; proofs?
- Semantic handles to communicate with other systems
- Avoiding duplication (*e.g.* additive magmas / multiplicative magmas)?

A.2.3.5. Have you gone in this direction? How did you represent the knowledge then?

The category framework for SageMath goes in this direction.

A.2.3.6. How do you collaborate on knowledge representation?

This is done through collaborative development of code, documentation and tests in the SageMath sources.

A.2.4. *Available software*

The SageMath library consists of 1.5 M lines of code (Python/Cython), and relies on hundreds of other software packages, in a myriad of languages.

The software, and particularly the category framework, builds on the available data and knowledge to construct a hierarchy of classes mirroring the categorical properties. Those are used for code factorization, documentation, and generic testing. For instance, a computation relying on the lattice of categories would be helped by the conclusion that if X is a division ring and X is a finite set, then X is a finite field.

A.3. **GAP**

A.3.1. *Overview at a high level of the* GAP *system*

GAP [**gap**] is an open-source system for computational discrete algebra, with particular emphasis on Computational Group Theory. GAP provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. It is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more.

A.3.2. *Available data*

GAP includes a number of data libraries listed online[4]. Some of them are part of the core GAP system, while some others belong to GAP packages. Their exact format may vary, but in all cases there are some text files with data and there is certain code responsible for processing particular pieces of information from those files. In some cases, the data library may only consist of the GAP code which will construct GAP objects on demand. Documentation is contained in the manual of the GAP system or relevant packages; however, it may not contain technical details which in the best case will be placed in README files or as comments in the code. Usually once produced, the data libraries are only changed when new data are added to them. Existing data may be altered only in case of discovered errors.

A.3.3. *What knowledge do you have?*

Apart from the knowledge that is stored in data libraries as explained above, there is a wealth of knowledge about properties of algebraic objects, or how to compute them, encoded in method installation and code. This knowledge can often not easily be extracted from the system.

However, the GAP type system has a number of advantages over a ”standard” object-oriented model for algebraic computation. Among the most important are:

- Method selection based equally on the types of all arguments. Thus, in implementing an extension field K of an existing field L, new methods for multiplying kl and lk can be added without any special support. Similarly, inheritance applies to all arguments equally.
- Method selection can take account of information accumulated during the lifetime of an object. For instance, as soon as a group is found to be abelian, special methods for abelian groups will be applied to it. Similarly, when the size of a group has been determined once, not only is it remembered in case it is needed again, but different methods for other computations may be selected to take account of this information.

A central idea in the design of GAP is that as much of possible of the core functionality should be polymorphic, so that it can be applied to any mathematical object with appropriate properties, without knowing the underlying representation. Thus if you create some new kind of GAP object, supply a method for multiplying such objects, and claim that it is associative, then you should be able to make semigroups from your objects. With additional methods and some additional claims of algebraic properties, you can make groups, rings or algebras.

A.3.4. *What software do you have?*

GAP has a kernel written in C. It implements:

- the GAP language,
- an interactive environment for developing and using GAP programs,
- memory management, and
- fast versions of time critical operations for various data types.

[4] http://www.GAP-system.org/Datalib/datalib.html

All the rest of the library of functions is written in the GAP language. Packages (user contributed extensions) are mainly written in the GAP language, but some also involve standalone executables. Some packages, for example, extend mathematical functionality of the system or add data libraries, while some others add infrastructural capabilities or links to other systems.

A.4. **LMFDB**

A.4.1. *Overview at a high level of the* LMFDB *system*

The *L-functions and Modular Forms DataBase* [**lmfdb**] aims to aggregate and integrate computational and mathematical knowledge about L-functions and other number theoretic objects, and to present these complex and interconnected objects reliably while maintaining accessibility. At a mathematical level, this could help provide a uniform view of the concept of L-function, objects which can (sometimes conjecturally) be produced out of very different mathematical constructions. The collaboration involves around 50 mathematicians of varying coding skills and with different mathematical expertise.

A.4.2. *Available data*

The entirety of the data held by the LMFDB is accessible through an API. One counts around 30 different types of objects stored, for a total of a few Tb. The data is downloadable directly.

The data is held in a MongoDB database server, holding around 30 or so databases, each with their own collections. It is held there as BSON (binary JSON), the internal format of Mongo documents.

Data that ends up in the LMFDB has many different origins. Some are historical computations. Most are done in either GAP, PARI, SageMath, MAGMA, etc, with the person who coded these original sources a member of the LMFDB who aims to make their data more accessible to their peers. Some of the data shown on the website is actually computed on the fly.

Data comes in through a variety of ad hoc ways, but essentially always transits through a JSON format before upload to the Mongo database. Updating is mostly done through some form of overwriting, but it is not uniform across the LMFDB. In the best cases, the data is stored completely separately from the LMFDB's own (Mongo) database, *e.g.* in a GitHub repository, under full revision control of text files, and there are scripts to populate the database from that. At some point there was discussion of allowing anyone to upload their data through an online form. This option has never been used seriously, and is not currently supported.

In general, proper referencing of data sources and documentation of its quality is a struggle, but there is recent improvement. Progress has been made on these through a new collection of "data quality" pages. The intention is to have source, extent and quality reliably documented for the main sections of the database.

In addition, the various formats are in the process of being formalised[5].

A.4.3. *What knowledge do you have?*

A.4.3.1. What are the sources of external knowledge?

Each participant in the LMFDB brings on specific mathematical knowledge about the objects studied, which might not be available to others in the collaboration. The LMFDB has the concept of *knowls*, which are encyclopaedic bits of content integrated alongside the data, and editable collaboratively. These help converge on common definitions of the objects described.

A.4.3.2. Can you point to implicit knowledge? Is it common knowledge?

There is a lot of implicit knowledge in the encodings chosen for the data (ad hoc formats and references), some of it is made explicit[6]. There is also a lot of implicit knowledge in the source

[5] See https://github.com/LMFDB/LMFDB-inventory, with the most advanced example (for elliptic curves) at https://github.com/LMFDB/LMFDB-inventory/blob/master/db-elliptic_curves.md. The formalisation format itself does not have a spec.

[6] emphe.g. at http://www.LMFDB.org/knowledge/show/ec.conductor_label

code. There is little common knowledge across the collaboration, or at least there is a lot that is not common.

A.4.3.3. What would you gain if it was made explicit/machine actionable?

The development process could probably be made more robust and efficient. The knowls currently serve as entry points for users and crucially also for onboarding future collaborators, as a stable basis for further collaboration. LMFDB could gain in productivity, robustness and ultimately utility if this process could be extended a bit further along the chain of contributions.

A.4.3.4. Have you gone in this direction? How did you represent the knowledge then?

The furthest the LMFDB has gone into the direction of formalising knowledge is in modularising as much as possible of the mathematical knowledge through knowls, creating an ad hoc ontology to classify them, and aligning it to the mathematical data objects that are presented. The LMFDB also tries to adhere to the concept of "one URL per object".

A.4.3.5. How do you collaborate on knowledge representation?

Edition of the knowls requires an account, which the LMFDB intends to offer to anyone who wishes to contribute. There is some versioning in place for knowls.

A.4.4. *What software do you have?*

The LMFDB is mostly written in Python, relies on SageMath and PARI/GP as libraries. It uses the database MongoDB (and possibly also an SQL one), uses the web framework Flask, and the templating engine Jinja.

A.4.4.1. What custom software are you running?

In a way SageMath is custom, since lots of LMFDB developers also contribute the relevant functionality to SageMath. Otherwise a whole lot of the logic is embedded in the website code.

A.4.4.2. How does the system use the data and the knowledge?

Generally, a URL path will be associated to a Jinja template, requiring simultaneous fetching of pre-entered knowledge (knowls, Mongo DB), precomputed data (Mongo DB), and on-the-fly computation based on this precomputed data or existing functions implemented in some of the Computer Algebra Software already used.

A.4.4.3. Which knowledge is implicit in the data you have?

A lot of information about the data encoding is implicit in the data itself. For instance, even if $[0, 4, 5, 1]$ is known to represent a polynomial, depending on the context it might represent $4 * x + 5 * x^2 + x^3$ or $x(x-4)(x-5)(x-1)$.

A.4.4.4. Which knowledge is implicit in the software you have?

When populating templates, some of the mathematical knowledge might be really entered through the code, by completing the template in different ways according to the calling class (*e.g.* elliptic curve L-functions are of degree 2).

I don't know if it is relevant here but we are also accumulating a set of bibliographic references and have already instituted a system for making citations within knowls very easy.

been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.