Towards a Unified Mathematical Data Infrastructure: Database and Interface Generation

Katja Berčič[1] [0000−0002−6678−8975], Michael Kohlhase[1] [0000−0002−9859−6337], and Florian Rabe[1,2] [0000−0003−3040−3655]

[1] FAU Erlangen-Nürnberg
[2] LRI Paris

Abstract. Data plays an increasing role in applied and even pure mathematics: datasets of concrete mathematical objects proliferate and increase in size, reaching up to $1\,TB$ of uncompressed data and millions of objects. Most of the datasets, especially the many smaller ones, are maintained and shared in an ad hoc manner. This approach, while easy to implement, suffers from scalability and sustainability problems as well as a lack of interoperability both among datasets and with computation systems.

In this paper we present another substantial step towards a unified infrastructure for mathematical data: a storage and sharing system with math-level APIs and UIs that makes the various collections findable, accessible, interoperable, and re-usable. Concretely, we provide a high-level data description framework from which database infrastructure and user interfaces can be generated automatically. We instantiate this infrastructure with several datasets previously collected by mathematicians. The infrastructure makes it relatively easy to add new datasets.

1 Introduction and Related Work

Motivation In general, mathematical objects are difficult to store scalably because of their rich and diverse structure. Usually, we have to focus on one aspect of mathematical objects in order to obtain representation languages that are simple enough to allow for maintenance in a scalable database. This is especially the case if we want to apply highly optimized general purpose technology like relational databases.

Mathematical data can be roughly separated into four main kinds.

1. *Symbolic data* focuses on formal expressions such as formulas, proofs, or programs. These are written in highly-structured formal languages with custom search and library management facilities.
2. *Narrative data* mixes natural language and formulas, which are usually presentation-oriented. These are written in, e.g., LaTeX, published as PDF or HTML files, and queries with text or metadata-based engines.

3. *Linked data* uses abstractions of mathematical objects that allow for optimized representations in knowledge graphs. These are stored in triple stores that allow for SPARQL-like querying.
4. Our interest here is on what we call **concrete data**, which encodes mathematical objects as simple data structures built from numbers, strings, lists, and records.

Mathematicians have constructed a wide variety of concrete datasets in terms of size, complexity, and domain. There is a vibrant and growing community of mathematicians that combine methods from experimental sciences with large scale computer support. An example of a prototypical experiment would be an algorithm that enumerates all objects of a certain kind (e.g., all finite groups up to order 2000), with measurements corresponding to computing properties (size, commutativity, normal subgroups, etc.) for each one.

While the objects themselves are typically abstract (e.g., an isomorphism class of finite groups) and involve hard-to-represent sets and functions, it is often possible to establish representation theorems that allow for unique characterization by a set of concrete properties. This allows storing even complex objects as simple data structures built from, e.g., integers, strings, lists, and records, which can be represented efficiently in standard relational databases. In the case of graphs, Brendan McKay supplies representation theorems and induced formats [McK] that are widely used to represent graphs as strings. This is typically done in combination with canonical labelings [BL83] to ensure that two graphs with different encodings are necessarily non-isomorphic. It is common to enumerate objects in order of increasing size and to let the algorithm run for as long as it scales, resulting in arbitrarily large datasets. It is not uncommon to e.g. encode the involved integers as strings because they are too big for the fixed width integer types of the underlying database.

Once the objects are stored in a database, they can be used for finding or refuting conjectures or simply as a reference table. One might want to ask questions like *"What are the Schläfli symbols of tight chiral polyhedra?* or *What regular polyhedra have a prime number of vertices"*. For complex hypothesis testing, the ideal solution would be an interface to a database from a computer algebra system or proof assistant system. This requires a non-trivial level of interoperability.

From the user perspective, looking up information about an object is most relevant when one does not have a lot of information about it. For example, it can be helpful to look up an object via some of its mathematical invariants. Another example is a search widget for the House of Graphs [Bri+13], which lets users draw the graph they are looking for.

State of the Art We used a living survey of mathematical databases as a market analysis. The survey consists of a table of datasets and collections of datasets [Bera] and accompanying (unstructured) information [Berb]. At the time of writing, about a hundred datasets are recorded in about 50 table entries. The datasets range from small, with up to 100 objects, to large, with $\approx 17 \cdot 10^9$ objects. Similarly varied is the authorship: from one author producing several smaller datasets, to FindStat [BSa14] with 69 contributors, LMFDB [LM] with 100 contributors, and the OEIS [OEIS] with thousands of contributors. Among these, the OEIS is the oldest and arguably most influential. It is notable for collecting not only mathematical objects but also symbolic data for their semantics, such as defining equations and generating functions. The most ambitious in scope in the LMFDB, a collection of mathematically related datasets that are conceptually tied together by Langland's program [Ber03].

Some databases like the OEIS and the LMFDB have become so important that substantial mathematical knowledge management (MKM) architecture has been developed for them. But most data sets are maintained in ad hoc systems such as text files with one entry per line or code that produces an array of objects for a computer algebra system. Less frequently, they are implemented as SQL databases or integrated with a computer algebra system such as SageMath, GAP, or Magma. There are multiple problems with this situation.

1. *Foundation/Documentation*: employed representation theorems can be difficult to find and apply, and there is usually no canonical representation.
2. *Database Encoding*: The data types typically available in the database cannot capture the richness of mathematical types. Thus, the original objects may be significantly different from their concrete encoding (e.g., a record of integers and booleans) that is stored in the database, and the database schema gives little information about the objects. Consequently, any query of the dataset requires not only understanding and reversing the database schema but also the possibly complex encoding.
3. *Silos*: Even if these are documented, any reuse of the dataset is tedious and error-prone in practice.

Thus, the current situation precludes developing MKM solutions that could provide, e.g., mathematical query languages (i.e., queries that abstract from the encoding), generic user interfaces, mathematical validation (e.g., type-checking relative to the mathematical types, not the database types), or cross-collection interlinking.

Vision/Goal To host all of these services and tools, we envision a universal unified infrastructure for mathematical data: a – possibly federated – storage and hosting system with math-level APIs and UIs that makes the various collections findable, accessible, and interoperable, and re-usable (see [FAIR18] for a discussion). We want to establish such a system (MathDataHub) as part of the MathHub portal [MH].

Contribution As a substantial step towards MathDataHub, we provide a high-level data description framework MDDL (Math Data Description Language), and a code generation system MBGen [MB]. Authors that describe their mathematical data collections in MDDL will be able to use MBGen to generate the necessary infrastructure extensions in MathDataHub, which can then host the data sets.

An MDDL schema consists of a set of property names with their mathematical types as well their concrete encodings in terms of low-level database types. This allows building user interface, validation, and querying tools that completely abstract from the encoding and the technicalities of the database system. The schemata are written as Mmt theories, which allows for concise formalizations of mathematical types.

The automated database setup process will be of benefit to *authors* of mathematical datasets by giving them better out-of-the-box tools and by making it easier to make more datasets available. In the long run, *users* of mathematical datasets will benefit from standardized interfaces and cross-database sharing and linking. Similarly, the community as a whole will benefit from novel methods such as machine learning applied to mathematical datasets or connecting areas by observing similarities between data from different areas.

Related Work The LMFDB collection of datasets enables authors to add their own datasets, cross-reference to existing datasets, and use a set of mature generic tools and a web-based user interface for accessing and searching their dataset. But it is limited to a specific topic, and does not systematically organize database schemata and encodings.

The House of Graphs portal is a home to a "searchable database of interesting graphs" as well as a repository, to which users can contribute their own datasets. Like the LMFDB, it is limited to a specific topic.

Some generic services exist for hosting research data, including systems like RADAR, publishers (Springer, albeit with a size limit), and EU-level infrastructures such as EUDAT and EOSC. But these have not been used extensively by mathematicians, partially due to a lack of awareness and a lack of added value they provide. Similarly unused go tools for managing and browsing general databases. These tend to be rich in features that are not useful from the perspective of mathematical datasets and have an initially steep learning curve, without features useful for mathematics at the end of it. We are somewhat surprised that even the simpler tools (based on the light-weight SQLite database, such as `sqliteonline.com`) appear to not be used much or at all – see [Bera].

The OpenDreamKit project developed the concepts of mathematical schemata and codecs that allow for systematically specifying the mathematical types in a dataset and their encodings [WKR17]. This provided the starting point for the present paper. These schemata and codecs used encodings as untyped JSON objects, and we had to extend them significantly for the present SQL-based approach.

The code generation system produces code for what is essentially an instance of a DiscreteZOO website. The DiscreteZOO project [BV18] itself is composed of a data repository, a website and a SageMath package. All three were developed for the use case of collections of graphs with a high degree of symmetry, but designed to be useful in a more general setting.

The MDDL Value Proposition Given a standard set of MDDL schemata and codecs for basic mathematical objects, authors mainly need to specify the construction of objects at the mathematical level. Arguably the mathematical level of description – though formal and detailed itself – is nearer to mathematical practice and to the computational systems used to generate the data set than database schemata. In particular, the pay-off of a MDDL specification is higher than from a database schema as the APIs and UIs, that would have to be written manually, are generated by MBGen. This difference becomes more pronounced as the mathematical objects become more complex and thus the differences between the mathematical and database structures grow.

Our work was partially motivated by a collaboration with the LMFDB community. The LMFDB system was based on a JSON database, where the use of schemata was optional. As a consequence, the mathematical meaning of the fields was often unclear to all but a few experts, creating severe FAIRness problems. These problems have since partially been resolved by a community-wide database schema/math documentation effort, which can be seen as an informal, human-oriented version of MDDL.

Overview In Section 2, we define our notion of a mathematical schema. In Section 3, we describe how to build a database instance from a set of schemata and show the user interface that we generate for the database. Section 4 concludes the paper and discusses future work.

Running Example We use the following – intentionally simple – scenario and dataset: Joe has collected a set of integer matrices together with their trace, eigenvalues, and the Boolean property whether they are orthogonal for his Ph.D. thesis. As he wants to extend, persist, and publicize this data set, he develops a MDDL description and submits it to MathDataHub. Jane, a collaborator of Joe's, sees his MathDataHub data set and is also interested in integer matrices, in particular their characteristic polynomials. To leverage and extend Joe's data set, she provides MDDL descriptions of her extension and further extends the MathDataHub matrix data infrastructure. Table 1 gives an overview of the situation.

Joe's dataset				Jane's column
M	$\mathrm{Tr}(M)$	Orthogonal	σ_M	$\det(\lambda I - M)$
$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$	2	yes	2, 1	$\lambda^2 - 3\lambda + 2$
$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$	4	no	3, 1	$\lambda^2 - 4\lambda + 3$
$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	0	yes	1, −1	$\lambda^2 - 1$

Table 1. Running Example: Joe's and Jane's Matrix Datasets

Acknowledgements The authors gratefully acknowledge helpful discussions with Tom Wiesing on MathHub integration and virtual theories. Discussions with Gabe Cunningham on concrete examples of questions a researcher might have and descriptions of tools that would help them helped shape our intuitions.

The work presented here was supported by EU grant Horizon 2020 ERI 676541 OpenDreamKit.

2 Defining Schema Theories

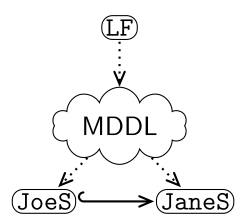
In this section, we give an overview of MDDL, the description language for datasets. Concretely, MDDL consists of a modular collection of MMT theories describing the mathematical and database types of a data set to be supported by the system, as well as the codecs that map between them. As such, MDDL forms a nucleus of a common vocabulary for typical mathematical datatypes. These can and should eventually be linked to representation standards in other domains. For mathematical data sets, the math-specific aspects attacked by our work are the dominant factor.

We avoid a formal introduction of MMT and refer to [Rab17; RK13] for details. For our purposes, only a small fragment of the language is needed and it is more appropriate to didactically introduce it by example.

Essentially, an MMT theory is a list of declarations $c : A$, where c is a name and A is the type of c. Additionally, an MMT theory T may have a *meta-theory* M. The basic idea is that M defines the language in which the expressions in T are written and the typing rules that determine which expressions are well-typed.

In our setup, there are three levels: The mathematical domain level and two meta-levels; see the diagram on the right.

1. the database schemata are the main theories of interest, here the MDDL specifications JoeS and JaneS provided by Joe and Jane.
2. their meta-theory is a collection of theories that define
   - a selection of typical mathematical datatypes
   - codecs for these datatypes to database types
3. the meta-theory of the MDDL theories is a variant of the logical framework LF.



LF already exists in MMT, whereas the MDDL theories were written specifically for our system. The schema theories are intended to be written by users for their respective databases or datasets. In essence, a schema theory defines one typed constant $c : A$ per database column and chooses the codec to use for that column. Table 2 gives an overview. The user describes the mathematical concepts relevant for the datasets (the first column) in the corresponding MMT language (second column). The MBGen system produces the database schema (third column) as well as a website interface for it.

In principle, MDDL could be written as a single theory once and for all. But in practice, it is important to make it user-extensible: as any mathematical type could occur in a database, MDDL will never be finished. Thus we employ a user-extensible theory graph, user schema theories can then import what they need. More precisely, the meta-theory of a schema theory must be some MMT theory that extends MDDL. In the long run, we expect MDDL to grow into a large, central, and continually growing theory graph that maximizes the integration

Mathematics	MMT	**DB**
sets of objects of the same type T	theory of T	table
object property c of type A	constant $c : A$ and choice of codec for A	column
object	model of theory	row

Table 2. Overview of concepts involved in schema theories

potential for users to exchange data across databases and computation systems. For this paper however, it is more instructive to use a small self-contained theory that allows describing the functionality in depth.

MDDL consists of multiple parts that we describe in the sequel:
– the mathematical types (theory `MathData`)
– the database types (`DbData`)
– the codecs translating between them (`Codecs`).

Along the way, we develop the schema theory for our running example. All of these are available online at [ST].

Example 1 (Joe's simplified schema theory).

Joe writes a schema theory `MatrixS` for matrix data using MDDL as the meta-theory – see the listing on the right. For a start, he just writes down the names of the columns and their types. Each declaration in the theory corresponds to one of the columns in Table 1: its name and type.

```
theory MatrixS: ?MDDL =
  ID: ℤ ❘
  mat: matrix ℤ 2 2 ❘
  trace : ℤ ❘
  orthogonal: bool ❘
  eigenvalues : list ℤ ❘
❚
```

We will show the finished schema theory with codecs for Joe's dataset in Ex. 2.

2.1 Mathematical Types and Operations

The theory MathData (see Listings 1.1 to 1.2) defines the mathematical types that we provide for building objects. These are unlimited in principle except that a mathematical type is only usable if at least one codec for it is available. Therefore, we only present a small representative set of examples.

It is important to understand that MathData is *not* a symbolic library of formalized or implemented mathematics in the style of proof assistants or computer algebra systems. There are three big differences, all of which make MathData much simpler:

1. We do not fix a logic or programming language as the foundation of the library. Foundational questions are almost entirely irrelevant for mathematical database schemata. MathData should be related to foundational libraries, but this is not a primary concern.

2. We do not include any axioms or theorems and proofs. It is reasonable to include some of those at a later stage, especially when increasingly complex types are used. Currently, their only purpose would be documentation – no system component performs any reasoning.
3. We do not define any of the types or operations. Formal definitions are irrelevant because they would never be applied anyway: all computation happens in the database.

Incidentally, because we do not use proofs or definitions, it is much easier to avoid fixing a foundation: most foundational complexity, e.g., induction, is not needed at all.

For similar reasons, it is neither necessary nor helpful to describe many operations on these types. Firstly, practical schema theories make little to no use of dependent types. Therefore, operations usually do not occur in schema theories, which only have to declare types. Secondly, practical datasets store primarily concrete data, where all operations have already been computed — irreducible symbolic expressions usually are not stored in the first place.

The only reasons why we declare any operations are *i*) to have constructors to build concrete objects, e.g., *nil* and *cons* for lists, *ii*) to use operations in queries to the extent that we can map them to corresponding database operations.

Literals We start with types for literals: Booleans, integers, strings, and UUIDs. The latter have no mathematical meaning but are often needed to uniquely identify objects in datasets. For each literal type, we provide the usual basic operations such as addition of integers.

Listing 1.1. Literals from MathData

```
bool : type ❙
int  : type ❘ # ℤ ❙
eq   : {a: type} a → a → bool ❘# 2 = 3❙
leq  : ℤ → ℤ → bool❘# 1 ≤2❙
geq  : ℤ → ℤ → bool❘# 1 ≥2❙
string : type❙
uuid: type❙
```

Collection Types We define a few standard collection types: $List\,A$, $Vector\,A\,n$, and $Option\,A$ are the usual types of arbitrary-length finite lists, fixed-length lists, and options containing objects of type A. We abbreviate matrices as vectors of vectors.

Algebraic Structures MMT theories can be naturally used to define types of algebraic structures such as groups, rings, etc. [MRK18]. Any such theory is immediately available as a type.

Listing 1.2. An excerpt from the MathData theory: collections

```
vector : type → ℤ → type ❘
  # vector 1 2 prec 10 ❙
empty : {a} vector a 0 ❙
single : {a} a → vector a 1 ❙
matrix : type → ℤ → ℤ → type ❘
  = [a,m,n] vector (vector a m) n ❙
option : type → type ❙
some : {a} a → option a ❙
none : {a} option a ❙
getOrElse : {a} option a → a → a ❙
```

Algebraic structures are difficult because they are complex, deeply structured objects. We omit the details here completely and only sketch a type of rings, which we need to build the type of polynomials.

2.2 Database Types and Operations

The theory DbData describes the most important types and operations provided by the target database, in our case PostgreSQL. This theory is fixed by PostgreSQL. However, we allow it to be extensible because PostgreSQL allows adding user-defined functions. That can be helpful to supply operations on encoded objects that match mathematical operations.

Listing 1.3. The DbData theory (simplified)

```
theory DbData : ur:?PLF =
  db_tp   : type ❙
  db_val  : db_tp → type | # V 1 prec −5 ❙
  db_null  : {a} V a ❙
  db_int , db_bool, db_string , db_uuid  : db_tp ❙
  db_array : db_tp → db_tp ❙
  eq  : {a} V a → V a → V db_bool |# 1 = 2 ❙ ...
❚
```

The types of DbData are signed 64 bit integers, double precision floating-point numbers, booleans, strings, and UUIDs as well as the null-value and (multidimensional) arrays of the above. The native PostgreSQL operations include the usual basic operations like Boolean operations and object comparisons.

2.3 Codecs: Encoding and Decoding Mathematical Objects

Overview The theory Codecs specifies encodings that can be used to translate between mathematical and database types. A **codec** consists of

- a mathematical type A, i.e., an expression in the theory MathData,
- a database type a, i.e., an expression in the theory DbData,
- partial functions that translate back and forth between MathData-expressions of type A and DbData-expressions of type a.

Obviously, decoding (from database type to mathematical type) is a partial function — for example, an encoding of integers as decimal-notation strings is not surjective. But encoding must be a partial function, too, because only *values* and not arbitrary expressions of type A can be encoded. For example, for integers, only the literals $0, 1, -1, \ldots$ can be encoded but not symbolic expressions like $1+1$, let alone open expressions like $1+x$. We do not define in general what a value is. Instead, every encoding function is partial, and we call *values* simply those expressions for which the encoding is defined.

Neither encoding nor decoding have to be injective. For example, a codec that encodes rational numbers as pairs of integer numerator and denominator might maximally cancel the fraction both during encoding and decoding. We also allow that provably equal expressions are encoded as different codes. That may be necessary for types with undecidable equality such as algebraic numbers. Similarly, we do not require that encoding *enc* and decoding *dec* are inverse functions. The only requirement we make on codecs is that for every expression e for which $enc(e)$ is defined, we have that $dec(enc(e))$ is defined and provably equal to e.

Like in [WKR17], we only *declare* but do not *implement* the codecs in MMT. It is possible in principle and desirable in the long run to implement the codecs in an soundness-guaranteeing system like a proof assistant and then generate codec implementations for various programming languages. But practical mathematical datasets for which our technology is intended often use highly optimized ad hoc encodings, where formal verification would make the project unnecessarily difficult early on. Instead, at this point we are content with a formal specification of the codec properties.

Contrary to [WKR17], we work with a *typed* database, which requires codecs to carry their database type a. Thus, we had to redevelop the notion of codecs completely.

An Initial Codec Library We only provide a representative set of codec examples here. For each literal type, we define an identity codec that encodes literals as themselves. In the case of integers, encoding is only defined for small enough integers. For large integers, we provide a simple codec that encodes integers using strings in decimal notation.

Apart from their intended semantics as codecs, the codec expressions are normal typed MMT expressions and therefore subject to binding and type-checking. For example, $ListAsArray$ is a codec operator of type

$$\Pi A : \mathsf{type}, a : db\_tp.\, codec\, A\, a \longrightarrow codec\,(list\, A)\,(db\_array\, a)$$

It takes a codec that encodes list elements of type A as database type a and returns a codec that encodes lists of type $list\, A$ as database arrays of type $db\_array\, a$. Similarly, we provide codec operators for all collection types.

There can be multiple codecs for the same mathematical type. The most well-known practical example where that matters is the choice between sparse and dense encodings of lists and related types such as univariate polynomials, matrices and graphs. But even basic types can have surprisingly many codecs as we have seen for integers above. For example, in the LMFDB collection, we found at least 3 different encodings of integers: the encoding is not obvious if integers are bigger than the bounded integers provided by the database.

In practice, we expect users to declare additional codecs themselves. This holds true especially when capturing legacy databases in our framework, where the existing data implicitly describes an ad-hoc encoding.

Choosing Codecs in Schema Theories Every column (constant) $c : A$ in a schema theory must carry a codec. This must be an expression $C : codec\, A\, a$ for some database type a.

Because the choice C of codec has no mathematical meaning, we do not make it part of the type of c. Thus, A is always just the mathematical type. Instead, we use MMT's metadata feature to attach C. Any MMT declaration can carry metadata, which is essentially a key-value list, where the key is an MMT identifier and the value an MMT expression.

We use the constant *codec* as the key and the codec C as the value. The resulting concrete syntax then becomes $c : A\ meta\ codec\ C$.

Example 2. Joe now adds codecs to his theory from Ex. 1 via metadata annotations as described above. Note that Joe does not need to add any new codecs, since MDDL already includes everything he needs. He also adds metadata `tag` annotations to specify databases schema and interface aspects. We will go into those in the next section.

Listing 1.4. Joe's schema theory with codecs

```
theory MatrixS : ?MDDL =
    mat: matrix ℤ 2 2 | meta ?Codecs?codec MatrixAsArray IntIdent |
        tag ?MDDL?opaque ❙
    trace : ℤ | meta ?Codecs?codec IntIdent ❙
    orthogonal: bool | meta ?Codecs?codec BoolIdent ❙
    eigenvalues : list ℤ | meta ?Codecs?codec ListAsArray IntIdent |
        tag ?MDDL?opaque ❙
❚
```

3 Database and Interface Generation

Given the MDDL framework presented above, the MBGen generator is rather straightforward. It uses the Slick library [Slk] for functional/relational mapping in Scala to abstract from the concrete database – we use Postgres SQL. MBGen creates the following Slick data model and uses it to create the actual database: For every schema theory T, MBGen generates a SQL table. The table name is the theory name, and it has a primary key called `ID`[3] of type `UUID`. For each declaration s in T, it generates a column, whose type is obtained from the codec declaration.

Example 3. When Joe runs MBGen on his schema theory from Listing 1.4, he obtains the database schema on the right. He can directly upload his data using the corresponding SQL `INSERT` statements, see the table below. Alternatively, he can use the math-level API provided by MBGen and specify the data at the mathematical level, i.e. as MMT declarations.

```
     Column     |   Type
----------------+----------
ID              | uuid
MAT             | integer[]
TRACE           | integer
ORTHOGONAL      | boolean
EIGENVALUES     | integer[]
Indexes: "MatrixS_pkey"
PRIMARY KEY, btree ("ID")
```

```
       ID          |     mat      | trace | orthogonal | eigenvalues
-------------------+--------------+-------+------------+-------------
e278b5e8-4404-...  | {2,0,0,1}    |     2 | t          | {2,1}
05a30ff0-4405-...  | {2,1,1,2}    |     4 | f          | {3,1}
1be3f022-4405-...  | {-1,0,0,1}   |     0 | t          | {1,-1}
```

Example 4 (Jane's extension). Jane specifies her dataset via a schema theory in Figure 1. She includes Joe's schema theory `MatrixS` and references the primary key of the corresponding database table as a foreign key. Jane has a slightly harder time importing her data set: she needs to obtain the the `ID` of the respective

[3] The data set might already have a pre-existing ID-like field, which is not a `UUID`. In this case we need to add a declaration for a custom index key.

matrices. Fortunately, this is only an easy SQL query away, since she is using the same matrix encoding.

```
theory MatrixWithCharacteristicS : ?SchemaLang =
  include ?MatrixS ❙
  matrixID: int ❘ meta ?SchemaLang?foreignKey ?MatrixS ❙
  characteristic : Polynomial IntegerRing ❘
    meta ?Codecs?codec PolynomialAsSparseArray IntIdent ❙
❚
```

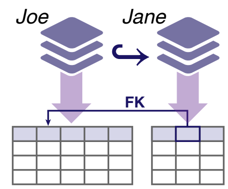


Fig. 1. Jane's Extensions for Matrices with Characteristic Polynomial

In addition to the codec, the user can provide further information about the dataset, so that the system can produce a better user interface. The tag

`collection`	Collection metadata, e.g. provenance data
`display`	Display name in the list of filters and in the results table head.
`hidden`	By default, hide in the results display.
`opaque`	Do not use for filtering.

Table 3. Metadata tags

`hidden` can be used when a dataset has many columns (like the graph datasets in DiscreteZOO). If there are hidden columns, the interface shows an additional widget, which lets a visitor choose which columns they want to see. If the tag `display` is not present, the column name is used for display. The tags particularly relevant for the interface the metadata tags and the tags `hidden` and `opaque`.

DiscreteZOO was already designed to work for general datasets. For the needs of MBGen, we further simplified the setup to the point where all dataset-specific information is contained in JSON files and can be hot-swapped. We also rewrote the frontend in React.JS for better performance and eventual integration into the React.JS-based MathHub front-end. The DiscreteZOO website interface was described in [BV18].

Evaluation: Mirroring Existing Databases To evaluate the setup, we have integrated the datasets currently hosted on the DiscreteZOO website in MathDataHub using the workflows described above. They are simpler than the much smaller running example: they only contain Boolean and integer valued properties and the objects are string-encoded. All the schema theories are available online [ST] in the folder `source`. The website stack obtained from these theories is equivalent to the original DiscreteZOO website. The links to the websites generated for the demos are available at the project wiki [MB].

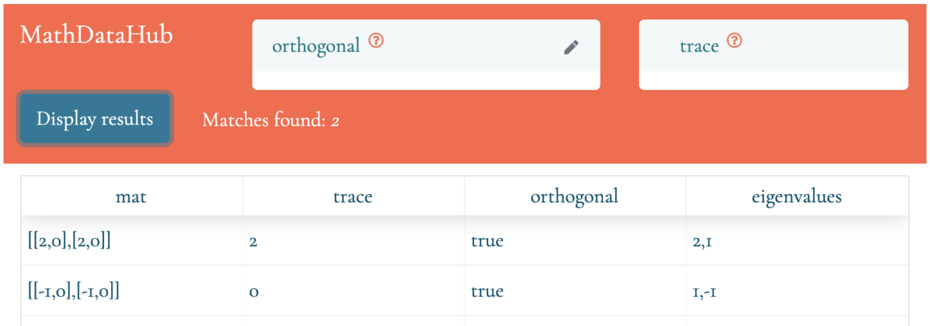


Fig. 2. Screenshot of the website for Joe and Jane's use case

This exercise shows that an author of a dataset with uncomplicated columns and no new codecs can start from an existing schema theory and adapt it to their needs in under an hour — essentially all that is required is to change the names of the columns.

4 Conclusion and Future Work

We have presented a high-level description language MDDL for mathematical data collections. MDDL combines storage aspects with mathematical aspects and relates them via an extensible collection of codecs. Dataset authors can specify schema aspects of their data sets and use the MBGen tool to generate or extend database tables and create customized user interfaces and universal mathematics-level APIs and thus tool stacks.

We have developed the framework and system to a state, where it already provides benefits to authors of smaller or simpler datasets, particularly students. The next step will be to stabilize and scale the MBGen system, fully integrate it into the MathHub system, and establish a public data infrastructure to the mathematical community. We will now discuss some aspects of future work.

Mirroring existing datasets Eventually, we hope that the system could be a useful mirror for the larger mathematical databases such as the FindStat, House of Graphs, LMFDB, and OEIS[4]. This would make (a significant subset) of the data available under a common, unified user interface, provide integrated, cross-library APIs and services like math-level search, and could bring out cross-library relations that were difficult to detect due to system borders. On the other hand, existing datasets act as goalposts for which features are needed most and thus drive MathDataHub development. Seeing which datasets can get realized in the system also serves as a measure of success.

[4] We can only hope to duplicate the generic parts of the user interface. Websites like the these four major mathematical database provide a lot of customized mathematical user functionality, which we cannot hope to reproduce generically.

A core library of codecs and support for common data types would lower the joining costs for MathDataHub by making MDDL schema theories straightforward to write.

Query Encoding In the online version of MathData, we also already specify commutativity conditions, which we omit here. Their purpose will be to specify when codecs are homomorphisms, i.e., when operations on mathematical values are correctly implemented by database operations on the encoded values. This will allow writing queries in user-friendly mathematical language, which are automatically translated to efficient executable queries in database language while guaranteeing correctness.

Extending the Data Model, User interface, and Query Languages by Provenance The MathDataHub system should provide a general model for data provenance, and propagate this to the UI and query levels. Provenance is an often-overlooked aspect of mathematical data, which can apply to data sets (like Joe's) single properties (e.g. Jane's characteristic polynomials), or even a single datum (e.g. the proof that the Monster group is simple). Provenance is the moral equivalent to proofs in pure mathematics, but is rarely recorded. It seems that the only chance to do this is by supporting this at the system level.

MathHub/MitM integration The schema theories rely on the representations of mathematical objects via their "mathematical types". These are supplied e.g. by the Math-in-the-Middle (MitM) Ontology or theorem prover libraries hosted on MathHub. Using these would make specifying MDDL schema theories even simpler, and would allow the MathDataHub UI to give access the mathematical definitions of the column values – e.g. by hovering over a name to get a definition in a tooltip.

Interoperability with Computer Algebra Systems Most mathematical data sets are computed and used in computer algebra systems. We have used the MitM framework [Koh+17] for distributed computation with the virtual theories underlying the MDDL framework [WKR17]. This needs to be reinstated for MathDataHub and scaled to the systems uses by the MathDataHub data sets.

Exploit MMT *modularity* MDDL is based on MMT, and we have already seen that MMT inclusions can directly be used to specify connected tables. We conjecture that MMT views correspond to database views; and would like to study how this can be exploited in the MDDL framework.

A flexible Caching/Re-computation Regime Instead of storing data, we can compute them on demand; similarly, computation can be done once and the data stored for later use. The better option comes down to the trade-off between computing and storage costs. This trade-off depends on hardware power and costs on one hand and community size and usage patterns on the other. It is generally preferable to perform computation "near" where the data is stored.

Mathematical Query Languages We have seen that the MDDL framework allows to directly derive a query interface for MathDataHub and note that this is inherently cross-dataset – a novel feature afforded by the modularity MDDL inherits from MMT. We conjecture that the framework allows the development of mathematical query languages – using e.g. [Rab12] as a starting point. The main research question is how to push computation to the database level (as opposed to the database model or the web client). The PostgreSQL database supports extensions by stored procedures. This would be one option for implementing additional filters for the website, such as enabling the condition "is prime" on integers. The stored procedures could also be used as a method for implementing "virtual columns", or columns that are computed, rather than stored.

References

[Bera] Katja Berčič. *Math Databases table.* URL: https://mathdb.mathhub.info/ (visited on 01/15/2019).

[Berb] Katja Berčič. *Math Databases wiki.* URL: https://github.com/MathHubInfo/Documentation/wiki/Math-Databases (visited on 01/15/2019).

[Ber03] Steve Bernstein Joseph Gelbart, ed. *An Introduction to the Langlands Program.* Birkhäuser, 2003. ISBN: 3-7643-3211-5.

[BKS17] Johannes Blömer, Temur Kutsia, and Dimitris Simos, eds. *MACIS 2017.* LNCS 10693. Springer Verlag, 2017.

[BL83] László Babai and Eugene M. Luks. "Canonical Labeling of Graphs". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing.* STOC '83. New York, NY, USA: ACM, 1983, pp. 171–183. ISBN: 0-89791-099-0. DOI: 10.1145/800061.808746.

[Bri+13] Gunnar Brinkmann et al. "House of Graphs: a database of interesting graphs". In: *Discrete Appl. Math.* 161.1-2 (2013), pp. 311–314. ISSN: 0166-218X. DOI: 10.1016/j.dam.2012.07.018. URL: https://doi.org/10.1016/j.dam.2012.07.018.

[BSa14] C. Berg, C. Stump, and al. *FindStat: The Combinatorial Statistic Finder.* http://www.FindStat.org. [Online; accessed 31 August 2016]. 2014.

[BV18] Katja Berčič and Janoš Vidali. "DiscreteZOO: a Fingerprint Database of Discrete Objects". 2018. URL: https://arxiv.org/pdf/1812.05921.pdf.

[FAIR18] European Commission Expert Group on FAIR Data. *Turning FAIR into reality.* 2018. DOI: 10.2777/1524. URL: https://doi.org/10.2777/1524.

[Koh+17] Michael Kohlhase et al. "Knowledge-Based Interoperability for Mathematical Software Systems". In: *MACIS 2017: Seventh International Conference on Mathematical Aspects of Computer and Information Sciences.* Ed. by Johannes Blömer, Temur Kutsia, and Dimitris Simos. LNCS 10693. Springer Verlag, 2017, pp. 195–210.

URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/MACIS17-interop/crc.pdf.
[LM] *The L-functions and Modular Forms Database*. URL: http://www.lmfdb.org (visited on 02/01/2016).
[MB] *MBGen description, links demos and code repositories*. URL: https://github.com/MathHubInfo/Documentation/wiki/MBGen (visited on 05/03/2019).
[McK] Brendan McKay. *Description of graph6, sparse6 and digraph6 encodings*. URL: http://users.cecs.anu.edu.au/~bdm/data/formats.txt.
[MH] *MathHub.info: Active Mathematics*. URL: http://mathhub.info (visited on 01/28/2014).
[MRK18] D. Müller, F. Rabe, and M. Kohlhase. "Theories as Types". In: *Automated Reasoning*. Ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Springer, 2018, pp. 575–590.
[OEIS] *The On-Line Encyclopedia of Integer Sequences*. URL: http://oeis.org (visited on 05/28/2017).
[Rab12] Florian Rabe. "A Query Language for Formal Mathematical Libraries". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 142–157. ISBN: 978-3-642-31373-8. arXiv: 1204.4685 [cs.LO].
[Rab17] Florian Rabe. "How to Identify, Translate, and Combine Logics?" In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.
[RK13] F. Rabe and M. Kohlhase. "A Scalable Module System". In: *Information and Computation* 230.1 (2013), pp. 1–54.
[Slk] *Slick, Functional Relational Mapping for Scala*. URL: http://slick.lightbend.com/ (visited on 03/16/2019).
[ST] Florian Rabe and Katja Berčič. *Schema theories repository for the prototyper*. URL: https://gl.mathhub.info/ODK/discretezoo (visited on 03/14/2019).
[WKR17] Tom Wiesing, Michael Kohlhase, and Florian Rabe. "Virtual Theories – A Uniform Interface to Mathematical Knowledge Bases". In: *MACIS 2017: Seventh International Conference on Mathematical Aspects of Computer and Information Sciences*. Ed. by Johannes Blömer, Temur Kutsia, and Dimitris Simos. LNCS 10693. Springer Verlag, 2017, pp. 243–257. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/MACIS17-vt/crc.pdf.