A Proposal for an OpenMath JSON Encoding OpenMath workshop CICM 2018

Michael Kohlhase Tom Wiesing

October 14, 2018

Abstract

OpenMath is a semantic representation of mathematical objects.

There are several encodings of OpenMath Objects, most notably the XML and Binary encodings. JSON is a lightweight data-interchange format that is present natively in many programming languages.

A few OpenMath JSON encodings already exist, which all have their advantages and disadvantages. These commonly correspond to a naive representation of the XML encoding and thus do not make use of some of the features that JSON offers.

In this paper we propose a new OpenMath JSON encoding, which combines the advantages of the above.

[1] [2] EdN:1 EdN:2

1 Introduction

OpenMath [3] is a semantic representation of mathematical objects. Because this paper is submitted to the OpenMath workshop, we will assume that reader is familar with OpenMath and will not introduce it further here. [3] EdN:3

JSON [7], short for **J**ava**S**cript **O**bject **N**otation, is a lightweight data-interchange format. While being a subset of JavaScript, it is defined independently. JSON can represent both primitive types and composite types.

[1] EdNote: Need to make introduction + conclusion flow better

[2] EdNote: Styling the paper/citation style?

[3] EdNote: Is this OK?

Primitive JSON types are strings (e.g. `"Hello␣world"`), Numbers (e.g. `42` or `3.14159265`), Booleans (`true` and `false`) and `null`. Composite JSON types are either (non-homogeneous) arrays (e.g. `[1, "two", false]`) or key-value pairs called objects (e.g. `{"foo": "bar", "answer": 42}`).

Constructs corresponding to JSON objects are found in most programming languages. Futhermore, the syntax is very simple; hence many languages have built-in facilities for translating their existing data structures to and from JSON. The use for an OpenMath JSON encoding is clear: It would enable easy use of OpenMath across many languages.

There are existing approaches for encoding OpenMath as JSON. We will discuss two particular ones here.

XML as JSON The JSONML standard[8] allows generic encoding of arbitrary XML as JSON. This can easily be adapted to the case of OpenMath. To encode an OpenMath object as JSON, one first encodes it as XML and then makes use of JSONML in a second step. Using this method, the term $\mathrm{plus}(x, 5)$ would correspond to:

```
[
    "OMOBJ",
    {"xmlns":"http://www.openmath.org/OpenMath"},
    [
        "OMA",
        [
            "OMS",
            {"cd": "arith1", "name": "plus"}
        ],
        [
            "OMV",
            {"name": "x"}
        ],
        [
            "OMI",
            "5"
        ]
    ]
]
```

This translation has the advantage that it is near-trivial to translate between the XML and JSON encodings of OpenMath. It also has some disadvantages:

- The encoding does not use the native JSON datatypes. One of the advantages of JSON is that it can encode most basic data types directly, without having to turn the data values into strings. To encode the floating point value `1e-10` (a valid JSON literal) using the JSONML encoding, one can not directly place it into the result. Instead, one has to turn it into a string first. Despite many JSON implementations providing such a functionality, in practice this would require frequent translation between strings and high-level datatypes. This is not what JSON is intended for, instead the provided data types should be used.

- The akwardness of some of the XML encoding remains. Due to the nature of XML the XML encoding sometimes needs to introduce elements that do not directly correspond to any OpenMath objects. For example, the *OMATP* element is used to encode a set of attribute / value pairs. This introduces unnecessary overhead into JSON, as an array of values could be used instead.

- Many languages use JSON-like structures to implement structured data types. Thus it stands to reason that an OpenMath JSON encoding should also provide a schema to allow languages to implement OpenMath easily. This is not the case for a JSONML encoding.

OpenMath-JS The openmath-js [4] encoding takes a different approach. It is an (incomplete) implementation of OpenMath in JavaScript and was developed by Nathan Carter for use with Lurch [5] on the web. It is written in literate coffee script, a derivative language of JavaScript.

In this encoding, the term plus$(x, 5)$ would correspond to:

```
{
  "t":"a",
  "c": [
    {
      "t":"sy",
      "cd":"arith1",
      "n":"plus"
    },
    {
      "t":"v",
```

```
        "n":"x"
      },
      {
        "t":"i",
        "v":"5"
      }
    ]
}
```

This encoding solves some of the disadvantages of the JSONML encoding, however it still has some drawbacks:

- It was written as a JavaScript, not JSON, encoding. The existing library provides JavaScript functions to encode OpenMath objects. However, the resulting JSON has only minimal names. This makes it difficult for humans to read and write directly.
- No formal schema exists, like in the JSONML encoding.

[4] Given these disadvantages, it makes sense to instead create a new OpenMath-JSON encoding. This encoding should combine the advantages of the above two. EdN:4

In particular, it should be close to the XML encoding, and at the same time make use of native JSON concepts. Furthermore, we want to formalize this encoding, which is not achieved by any existing approach, and thus enable easy validation.

2 The OpenMath-JSON encoding

To formalize our encoding, we decided to use JSON Schema [2]. This defines a vocabulary allowing us to validate and annotate JSON documents. Additionally, tools for programatic verification exist in many languages.

Unfortunatly, JSON schema is often tedious to write and read for humans. This is especially true when it comes to recursivly defined data strucutures. OpenMath has many recusrive structures. Instead of writing our encoding in JSON Schema directly, we decided to write the schema in a different language and then compile it to JSON Schema.

[4] EdNote: Potientally re-formulate this

For this purpose, we decided to make use of TypeScript [9]. TypeScript is a language derived from JavaScript – TypeScript files are JavaScript plus type annotations. As such, it can be easily written and understood by humans. On top of typescript, we make use of a compiler [1] from TypeScript definitions into JSON Schema.

2.1 Encoding Details

In general, objects in our encoding look similar to the following:

```
{
    "kind": "OMV",
    "id": "something",
    "name": "x"
}
```

The `kind` attribute specifies which kind of OpenMath object this is. These values correspond to the element names used in the XML encoding. This correspondence lays the foundations of easy translation between the two. In TypeScript this property is also refered to as a *Type Guard*, because if *guards* the type of object that is represented.

Like in the XML encoding it is possible to make use of structure sharing. For this purpose the `id` attribute can be used. We will come back to this in more detail below, when we define to the `OMR` type.

In the following we will go over the details of our encoding. For this we will make use of a TypeScript-like syntax, that is easily readable. In our description we omit the `id` attribute, which can be added to any encoded object. The complete source code of our encoding – and details on how to use it – can be found on Github [10].

2.1.1 Object Constructor – OMOBJ

The OpenMath Object Constructor – `OMOBJ` – is defined as follows:

```
{
    "kind": "OMOBJ",
    /** optional version of openmath being used */
    "openmath": "2.0",
    /** the actual object */
    "object": omel /* any element */
}
```

Concretly, the integer 3 encapsulated in an object constructor using this encoding is as follows:

```
{
    "kind": "OMOBJ",
    "openmath": "2.0",
    "object": {
        "kind": "OMI",
        "integer": 3
    }
}
```

Let us have a look at this first example attribute for attribute.

The first attribute – `kind` – represents the type of OpenMath object in question. Notice that it occurs twice – once in the `OMOBJ` and a second time in the wrapped `OMI`. We will talk in detail about integer representation below, and hence only care about this first one.

The second attribute – `openmath` – is defined as optional by our schema. This indicates the version of OpenMath that is being used – `"2.0"` in our case.

The third and final attribute is the `object` attribute. This contains the wrapped object – it is defined as of `omel` type. [5] This type `omel` can contain any OpenMath element – concretly primitive objects (Symbols `OMS`, Variables `OMV`, Integers `OMI`, Floats `OMF`, Bytes `OMB`, Strings `OMSTR`), complex elements (Application `OMA`, Attribution `OMATTR`, Binding `OMBIND`) or Errors `OME` and References `OMR`. In this particular case, we just have the integer 3. EdN:5

2.1.2 Symbols – OMS

An OpenMath Symbol is encoded as follows:

```
{
    "kind": "OMS",
    /** the base for the cd, optional */
    "cdbase": uri, /* any valid URI */,
    /** content dictonary the symbol is in, any uri */
    "cd": uri,
    /** name of the symbol */
    "name": name /* any valid symbol name */
}
```

[5] EDNOTE: Make sure the document is in this order?

Notice the `uri` and `name` types in the definition. These are not directly JSON types. We define the `uri` type to be a any JSON string that represents a valid URI. Similarly, we define the `name` type to be any JSON string that represents a valid symbol name.

For example to encode the `sin` symbol from the `transc1` CD:

```
{
    "kind": "OMS",
    "cd": "transc1",
    "name": "sin"
}
```

2.1.3 Variables – OMV

An OpenMath Variable is encoded as follows:

```
{
    "kind": "OMV",
    /** name of the variable */
    "name": name
}
```

We again make use of the `name` type here.

For example to encode a variable x:

```
{
    "kind": "OMV",
    "name": "x"
}
```

2.1.4 Integers – OMI

Unlike the previous elements, our encoding allows integers to be encoded in three different ways. In particular, we define them as follows:

```
{
    "kind": "OMI",
    //
    // exactly one of the following
    //

    /* any json integer */
```

```
    "integer": integer,
    /* any string matching ^-?[0-9]+$ */
    "decimal": decimalInteger,
    /* any string matching ^-?x[0-9A-F]+.$ */
    "hexadecimal": hexInteger
}
```

We allows integers to be represented as one of the following:

JSON Integers Representing an OpenMath Integer as a JSON integer allows making use of datatypes that JSON offers. This was one of the goals we wanted to achieve with our encoding.

JSON has no integer type in and of itself – it only provides a `number` type. To work around this in our schema, we define a custom integer type as any number that is an integer.

OpenMath integers can be of arbitrary size. While the JSON specification does not limit the size of numbers, it also allows any implementation the freedom to pick some limit. Thus for reasonably sized integers, this representation works well. But for larger numbers, the other two variants should be used.

Decimal Strings A second way to encode an OpenMath Integer is to make use of the straightforward decimal string encoding. This is any string consisting only of digits (and potentially having a minus sign in the case of a negative integer).

Hexademical String We can allow to encode an OpenMath Integer as a hexadecimal integer. To make this different from decimmally encoded integers, we require a lower-case x to be placed before the string. This also has the advantage that this corresponds exactly to the XML encoding.

We can thus encode the integer -120 in three different ways:

- as a JSON integer

  ```
  {
      "kind": "OMI",
      "integer": -120
  }
  ```

- as a decimal-encoded string

```
{
    "kind": "OMI",
    "decimal": "-120"
}
```

- as a hexadecimal-encoded string

```
{
    "kind": "OMI",
    "hexadecimal": "-x78"
}
```

2.1.5 Floats – OMF

Like integers, we allow floats to be encoded in three different ways:

```
{
    "kind": "OMF",

    //
    // exactly one of the following
    //

    /* any json number */
    "float": number,
    /* any string matching
        (-?)([0-9]+)?(\.[0-9]+)?([eE](-?)[0-9]+)? */
    "decimal": decimalFloat,
    /* any string matching ^([0-9A-F]+)$ */
    "hexadecimal": hexFloat
}
```

Here, the different cases work exactly like in the integer case.

We can represent a float either as a native JSON number, a string in decimal representation or a string in hexadecimal representation. Just like above, the string representations correspond to the ones allowed by the XML encoding.

For example the floating point number 10^{-10} can be represented in three different ways:

- as a JSON float

  ```
  {
      "kind": "OMF",
      "float": 1e-10
  }
  ```

- as a decimal-encoded string

  ```
  {
      "kind": "OMF",
      "decimal": "0.0000000001"
  }
  ```

- as a hexadecimal-encoded string

  ```
  {
      "kind": "OMF",
      "hexaecimal": "3DDB7CDFD9D7BDBB"
  }
  ```

2.1.6 Bytes – OMB

Bytes can be encoded in two different ways:

```
{
    "kind": "OMB",
    //
    // exactly one of the following
    //

    /** an array of bytes
       where a byte is an integer from 0 to 255 */
    "bytes": byte[],
    /** a base64 encoded string */
    "base64": base64string
}
```

Array of bytes This encoding again makes use of JSON datastructures – representing bytes as a concrete list of bytes. As the byte datatype does not exist directly in JSON, we represent a single byte as an integer between 0 and 255 (inclusive).

Base64 encoded string A base64-encoded string of bytes corresponds to the XML encoding.

For example we can encode the ascii bytes of the string *hello world*

- as a byte array

```
{
    "kind": "OMB",
    "bytes": [
        104, 101, 108, 108, 111, 32,
        119, 111, 114, 108, 100
    ]
}
```

- as a base64-encoded string

```
{
    "kind": "OMB",
    "base64": "aGVsbG8gd29ybGQ="
}
```

2.1.7 Strings – OMS

The encoding of strings is straightforward – we can make use of native JSON strings.

```
{
    "kind": "OMSTR",
    /** the string */
    "string": string
}
```

Thus the string `"Hello␣world"` is encoded as follows:

```
{
    "kind": "OMSTR",
    "string": "Hello␣world"
}
```

2.2 The Demo Site

To demonstrate our OpenMath-JSON encoding, we have created a demo site which can be found at [11]. This site is implemented in TypeScript and encapsulated using Docker[6].

The demo site serves three purposes.

Primarily, it serves as a presentation of the encoding, providing examples and documenting it's usage.

Secondly, it enables validation of OpenMath JSON objects. This can be seen in Figure[6]. The user can enter some JSON, press the *Validate JSON* button, and receive immediate feedback if their JSON is a valid OpenMath object or not. In particular, the user can also see a detailed error message if their object is not valid OpenMath JSON. EdN:6

This makes use of the OpenMath JSON schema, and validates the users' JSON using a generic JSON Schema Validator. Furthermore, this is also exposed using a REST API, enabling easy validation of OpenMath JSON in other applications.

Thirdly, it enables translation between XML and JSON OpenMath objects. Like for validation, the site enables the user to enter some JSON and be presented with some XML and vice-versa. This can be seen in Figure[7]. EdN:7

As we designed our encoding with this translatability goal in mind, the implementation of it was straight-forward. Nonetheless, this translation is also exposed using a REST API.

3 Conclusion

In this paper we have established that an OpenMath JSON encoding enables using OpenMath in many programming languages. The existing approaches for such an encoding did not make use of many of the native JSON features, hence we have developed our own encoding. This encoding is both easily translatable to and from the JSON encoding and makes use of native JSON features.

[8] EdN:8

[6] EdNote: Make figure

[7] EdNote: Make figure + screenshot

[8] EdNote: Not sure what else to write here

References

[1] Dominik Moritz et al. *ts-json-schema-generator*. Accessed: 2018-10-14.

[2] H. Andrews and A. Wright. *JSON Schema: A Media Type for Describing JSON Documents*. Tech. rep. Internet Engineering Task Force (IETF), 2018. URL: `https://tools.ietf.org/html/draft-handrews-json-schema-01`.

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

[4] Nathan C. Carter. *openmath-js*. Accessed: 2018-10-13.

[5] Nathan C. Carter and Kenneth G. Monks. "Lurch: a word processor built on OpenMath that can check mathematical reasoning". In: *Joint Proceedings of the MathUI, OpenMath, PLMMS, and ThEdu Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics 2013*. (Bath, UK, July 8–12, 2013). Ed. by Christoph Lange et al. CEUR Workshop Proceedings 1010. Aachen, 2013. URL: `http://ceur-ws.org/Vol-1010/paper-23.pdf`.

[6] Docker Inc. *Docker - Build, Ship, and Run Any App, Anywhere*. Accessed: 2018-10-14.

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

[8] Stephen M. McKamey. *JSON Markup Language*. Accessed: 2018-10-13.

[9] Microsoft. *TypeScript - JavaScript that scales.* Accessed: 2018-10-14.

[10] *tkw1536/OpenMath-JSON*. GitHub repository at `https://github.com/tkw1536/OpenMath-JSON`. accessed 2018-10-14. URL: `https://github.com/tkw1536/OpenMath-JSON`.

[11] Tom Wiesing. *OpenMath-JSON*. URL: `https://omjson.kwarc.info/` (visited on 10/13/2018).