A Proposal for an OpenMath JSON Encoding

Tom Wiesing    Michael Kohlhase

August 13, 2018
Recap: The OpenMath Information Architecture

- OpenMath provides extensible standards for representing the semantics of mathematical objects and communicating them between software systems.
- The main encoding is based on XML
  - XML has a convenient tree model
  - XML is supported in many programming languages

BUT: XML is not the only game in town.

Some communities/frameworks prefer the more lightweight JSON (especially in the Web services context).

OpenMath may want to provide a JSON encoding to cater to these communities.
Recap: The OpenMath Information Architecture

- OpenMath provides extensible standards for representing the semantics of mathematical objects and communicating them between software systems.
- The main encoding is based on XML
  - XML has a convenient tree model
  - XML is supported in many programming languages
- **BUT**: XML is not the only game in town.
- some communities/frameworks prefer the more lightweight JSON (especially in the Web services context)
- OpenMath may want to provide a JSON encoding to cater to these communities
What is JSON?

- JSON = **JavaScript Object Notation**
  - lightweight data-interchange format
  - subset of JavaScript (used a lot on the web)
  - defined independently

- **Primitive types**
  - Strings (e.g. "Hello world")
  - Numbers (e.g. 42 or 3.14159265)
  - Booleans (true and false)
  - null

- **Composite types**
  - Arrays (e.g. [1, "two", false])
  - Objects (e.g. {"foo": "bar", "answer": 42})
Why an OpenMath encoding for JSON?

- an OpenMath JSON encoding would make it easy to use across many languages
  - JSON support exists in most modern programming languages
    - corresponding native types common
    - serialization to/from JSON without external library
- some existing approaches for an OpenMath JSON encoding
  - discussed / suggested on the OpenMath mailing list
  - we will look at two examples here
**XML as JSON**

- **Idea:** Generically encode XML as JSON
- use the JSONML standard for this
- e.g. `plus(x,5)` corresponds to:

```json
[  
  "OMOBJ",  
  {"xmlns":"http://www.openmath.org/OpenMath"},  
  [  
    "OMA",  
    ["OMS", {"cd": "arith1", "name": "plus"}]],  
    ["OMV", {"name": "x"}],  
    ["OMI", "5"]  
  ]
]
```
XML as JSON (2)

- Advantages
  - based on well-known XML encoding
  - easy to understand based on it
- does not make use of JSON structures
  - all attributes are encoded as strings, even numbers
  - e.g. 1e-10 (a valid JSON literal) can not be used
- retains some of the XML awkwardness
  - introduces unnecessary overhead
  - e.g. some pseudo-elements (such as OMATP) are needed
OpenMath-JS

- OpenMath-JS
  - an (incomplete) implementation of OpenMath in JavaScript
  - developed by Nathan Carter for use with Lurch Math on the web
  - written in literate coffee script, a derivative language of JavaScript
- e.g. `plus(x, 5)` corresponds to:

```javascript
{
  "t": "a",
  "c": [
    {
      "t": "sy", "cd": "arith1", "n": "plus"},
    {
      "t": "v", "n": "x"},
    {
      "t": "i", "v": "5"
    }
  ]
}
```
OpenMath-JS (2)

- does make use of JSON native structures
  - much better than JSON-ML
  - small property names keep size of transmitted objects small
- comes with some problems
  - hard to read for humans
  - written for JavaScript, not JSON
  - no formal schema
Towards an OpenMath JSON Formalization

- we need to write a new OpenMath JSON encoding
  - combine advantages of the above two
  - should be close to the XML encoding
  - should make use of JSON concepts
- we want to formalize this JSON encoding
  - to verify JSON objects
  - not done by existing approaches
- comes with some positive side effects
  - formalization of JSON $\Rightarrow$ structure definition in most languages
  - trivial to use advanced serialization tools
    - e.g. Protocol Buffers, ZeroMQ
- we can use JSON Schema
  - a vocabulary allowing us to validate and annotate JSON documents
  - tools for verification exist
Towards an OpenMath JSON Formalization (2)

- JSON schema is often tedious to write and read
  - especially when it comes to recursive data types
  - but implementation of it still exist
- **Idea:** Write schema in a TypeScript, compile into a JSON schema
  - TypeScript = JavaScript + Type Annotations
  - easily writeable and understandable
  - a compiler from TypeScript Definitions into JSON Schema exists
- We have done this, and will present some examples in the following slides
Towards an OpenMath JSON Formalization (3)

- Wrote a JSON Schema
  - was written as described above
  - we will give an overview how this looks below
- Wrote a translator from OpenMath XML to JSON (we have actually built two)
  1. web demo on (https://omjson.kwarc.info)
  2. as part of MMT (i.e. Scala) in the form of a RESTful API
General Structure of OpenMath objects

- represent each OM Object as a Hashmap:
  
  ```
  {
    "kind": "OMV",
    "id": "something",
    "name": "x"
  }
  ```

- **kind** attribute specifies the type
  - called a *type guard* in TypeScript
  - has the same names as elements in the XML encoding

- **id** attribute used for structure sharing
  - like in xml
  - referenced using OMR kind (we will come back to this later)

- the examples
  - use TypeScript syntax (easily readable)
  - omit the **id** attribute
Object Constructor - OMOBJ

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

▶ e.g. the number 3

{ "kind": "OMOBJ", "openmath": "2.0", "object": { "kind": "OMI", "integer": 3 } } }
Symbols - OMS

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

▶ e.g. the sin symbol from the transc1 CD

{  
"kind": "OMS",  
"cd": "transc1",  
"name": "sin"
}
Variables - OMV

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

▶ e.g. the variable $x$  

{  
"kind": "OMV",  
"name": "x"  
}
Integers - OMI (1)

- integers can be represented in three ways
  - as a native JSON integer
  - as a decimal-encoded string (like in XML)
  - as a hexadecimal-encoded string (like in XML)

```json
{
    "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
}
```
Integers - OMI (2)

- e.g. \(-120\) represented in three ways:
  - as a JSON integer
    
    ```json
    {
      "kind": "OMI",
      "integer": \(-120\)
    }
    ```
  
  - as a decimal-encoded string
    
    ```json
    {
      "kind": "OMI",
      "decimal": \("-120\"
    }
    ```
  
  - as a hexadecimal-encoded string
    
    ```json
    {
      "kind": "OMI",
      "hexadecimal": \("-x78\"
    }
    ```
Floats - OMF (1)

- floats can also be represented in three ways
  - as a native JSON number
  - using their decimal encoding (like in XML)
  - using their hexadecimal encoding (like in XML)

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

  //
  // exactly one of the following
  //

  /* any json number */
  "float": float,

  /* any string matching */
  /* any string matching */
  /* any string matching */
  /* any string matching */
  /* any string matching */

  "decimal": decimalFloat,

  /* any string matching */

  "hexadecimal": hexFloat
}
```
Floats - OMF (2)

- e.g. $10^{-10}$ represented in three ways:
  - as a JSON float
    ```json
    {
      "kind": "OMF",
      "float": 1e-10
    }
    ```
  - as a decimal-encoded string
    ```json
    {
      "kind": "OMF",
      "decimal": "0.0000000001"
    }
    ```
  - as a hexadecimal-encoded string
    ```json
    {
      "kind": "OMF",
      "hexadecimal": "3DDB7CDFD9D7BDBB"
    }
    ```
bytes can be represented in two ways

- as an array of bytes
- as a string encoded in base64

```json
{
    "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
}
```
Bytes - OMB (2)

▶ e.g. the ascii bytes of *hello world* represented in two ways:
▶ as a byte array

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

▶ as a base64-encoded string

```json
{
    "kind": "OMB",
    "base64": "aGVsbG8gd29ybGQ="
}
```
Strings - OMSTR

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

e.g. {
    "kind": "OMSTR",
    "string": "Hello
    world"
}]
```
Applications - OMA (1)

```json
{  
  "kind": "OMA",
  /** the base for the cd, optional */  
  "cdbase": uri,
  /** the term that is being applied */  
  "applicant": omel,
  /** the arguments that the applicant is being applied to. Optional and assumed to be empty if omitted */  
  "arguments"?: omel[]
}
```
Applications - OMA (2)

► e.g. $\sin(x)$

```json
{
    "kind": "OMA",
    "applicant": {
        "kind": "OMS",
        "cd": "transcl",
        "name": "sin"
    },
    "arguments": [{
        "kind": "OMV",
        "name": "x"
    }]
}
```
Attributions - OMATTR (1)

```json
{
    "kind": "OMATTR",
    /** the base for the cd, optional */
    "cdbase": uri,
    /** attributes attributed to this object, non-empty */
    "attributes": ([
        OMS, omel|OMFOREIGN
    ])
    /** object that is being attributed */
    "object": omel
}
```

- attributes are represented as an array of pairs containing
  - the name of the attribute
  - the value of the attribute
Attributions - OMATTR (2)

- e.g. to annotate a variable $x$ as having a real type

```json
{
  "kind": "OMATTR",
  "attributes": [
    [
      {
        "kind": "OMS", "cd": "ecc", "name": "type"
      },
      {
        "kind": "OMS", "cd": "ecc", "name": "real"
      }
    ]
  ],
  "object": {
    "kind": "OMV",
    "name": "x"
  }
}
```
variables being attributed are represented as a list with each element either
  ▶ an OMV variable
  ▶ an OMATTR where the attributed object is a variable (attvar)
e.g. $\lambda x. \sin(x)$

```json
{
  "kind": "OMBIND",
  "binder": {
    "kind": "OMS", "cd": "fns1", "name": "lambda" },
  "variables": [
    { "kind": "OMV", "name": "x" }
  ],
  "object": {
    "kind": "OMA",
    "applicant": {
      "kind": "OMS", "cd": "transc1", "name": "sin" },
    "arguments": [
      { "kind": "OMV", "name": "x" }
    ]
  }
}
```
Errors - OME (1)

```json
{
  "kind": "OME",
  /** the error that has occurred */
  "error": OME,
  /** arguments to the error, optional */
  "arguments"?: (omel|OMFOREIGN)[
}
```
e.g. to annotate a *division by zero* error in $x/0$

```json
{
    "kind": "OME",
    "error":
        {
            "kind": "OMS", "cd": "aritherror",
            "name": "DivisionByZero" 
        },
    "arguments": [[
        "kind": "OMA",
        "applicant": {
            "kind": "OMS", "cd": "arith1",
            "name": "divide" 
        },
        "arguments": [
            { "kind": "OMV", "name": "x" },
            { "kind": "OMI", "integer": 0 }
        ]
    ]]
}
```
Foreign Objects - OMFOREIGN

- { "kind": "OMFOREIGN",
  /** encoding of the foreign object, optional */
  "encoding"?: string,
  /** the foreign object */
  "foreign": any
}

- e.g. to represent a \LaTeX\ math term \( \sin(x) \)

  { "kind": "OMFOREIGN",
    "encoding": "text/x-latex",
    "foreign": "$\sin(x)$" 
  }
we can reference any object with an id

```json
{
    "kind": "OMR"
    /** element that is being referenced */
    "href": uri
}
```

e.g. term $f(f(f(a, a), f(a, a)), f(f(a, a), f(a, a)))$ encoded as a DAG

```latex
\begin{array}{c}
  f \quad f \quad f \quad a \\
\end{array}
```
encoded as $f(f(f(a, a), y), x)$
References - OMR (2)

```json
{
    "kind": "OMOBJ",
    "object": {
        "kind": "OMA",
        "applicant": {
            "kind": "OMV",
            "name": "f"
        },
        "arguments": [
            {
                "kind": "OMA",
                "id": "x",
                "applicant": {
                    "kind": "OMV",
                    "name": "f"
                },
                "arguments": [
                    {
                        "kind": "OMA",
                        "id": "y",
                        "applicant": {
                            "kind": "OMV",
                            "name": "f"
                        },
                        "arguments": [
                            {
                                "kind": "OMV",
                                "name": "a"
                            },
                            {
                                "kind": "OMV",
                                "name": "a"
                            }
                        ],
                        "href": "#y"
                    },
                    {
                        "kind": "OMR",
                        "href": "#x"
                    }
                ]
            }
        ]
    }
}
```
Summary

- Developed a JSON encoding for OpenMath Objects
- An OpenMath JSON encoding makes using OM much easier in many languages
  - most languages have structured data types built in
  - serialization into/from JSON exists natively in many languages
- existing approaches had disadvantages \( \rightarrow \) develop our own
  - simple to translate to/from the XML Encoding (see https://omjson.kwarc.info/)
  - uses JSON-native data types
- Future Work: Fully integrate into OpenMath ecosystem
  - what about JSON-based CD files? (I think rather not)
  - what about SCSCP?
    - easy to make use of Protocol Buffers or ZeroMQ based on this work
- Questions, Comments, Concerns?
Summary

- Developed a JSON encoding for OpenMath Objects
- An OpenMath JSON encoding makes using OM much easier in many languages
  - most languages have structured data types built in
  - serialization into/from JSON exists natively in many languages
- existing approaches had disadvantages \(\rightsquigarrow\) develop our own
  - simple to translate to/from the XML Encoding (see https://omjson.kwarc.info/)
  - uses JSON-native data types

- We propose to adopt this encoding and publish from the OM Web Site \(\rightsquigarrow\) OM Business Meeting

Future Work: Fully integrate into OpenMath ecosystem
- what about JSON-based CD files? (I think rather not)
- what about SCSCP?
  - easy to make use of Protocol Buffers or ZeroMQ based on this work

Questions, Comments, Concerns?
Summary

- Developed a JSON encoding for OpenMath Objects
- An OpenMath JSON encoding makes using OM much easier in many languages
  - most languages have structured data types built in
  - serialization into/from JSON exists natively in many languages
- existing approaches had disadvantages \( \rightarrow \) develop our own
  - simple to translate to/from the XML Encoding (see https://omjson.kwarc.info/)
  - uses JSON-native data types

- We propose to adopt this encoding and publish from the OM Web Site \( \rightarrow \) OM Business Meeting

- Future Work: Fully integrate into OpenMath ecosystem
  - what about JSON-based CD files? \( \rightarrow \) I think rather not
  - what about SCSCP?
  - easy to make use of Protocol Buffers or ZeroMQ based on this work

- Questions, Comments, Concerns?