A Proposal for an OpenMath JSON Encoding

Tom Wiesing   Michael Kohlhase

August 7, 2018

What is JSON?

- JSON = **J**ava**S**cript **O**bject **N**otation
  - 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. $\mathrm{plus}(x, 5)$ corresponds to:

```
[
    "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 akwardness
    - 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. $\mathrm{plus}(x, 5)$ corresponds to:

```
{
    "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 recusrive 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)

- ▶ We wrote a JSON Schema
  - ▶ was written as described above
  - ▶ we will give an overview how this looks below
- ▶ We also wrote a translator from OpenMath XML to JSON
  - ▶ a RESTful interface as part of MMT
  - ▶ was quick to implement given an existing XML implementation

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 dictonary 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)

```
{
    "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

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

  - as a decimal-encoded string

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

  - as a hexadecimal-encoded string

    ```
    {
        "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)

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

    //
    // exactly one of the following
    //

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

Floats - OMF (2)

- e.g. 10^{-10} represented in three 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"
    }
    ```

Bytes - OMB (1)

- bytes can be represented in two ways
  - as an array of bytes
  - as a string encoded in base64

```
{
    "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

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

  - as a base64-encoded string

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

Strings - OMSTR

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

- e.g.
  ```
  {
      "kind": "OMSTR",
      "string": "Hello␣world"
  }
  ```

Applications - OMA (1)

- ```
  {
      "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. $\sin(x)$

```
{
    "kind": "OMA",
    "applicant": {
        "kind": "OMS",
        "cd": "transc1",
        "name": "sin"
    },
    "arguments": [{
        "kind": "OMV",
        "name": "x"
    }]
}
```

Attributions - OMATTR (1)

- ▶ ```
  {
      "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

```
{
    "kind": "OMATTR",
    "attributes": [
        [
            { "kind": "OMS", "cd": "ecc", "name": "type" },
            { "kind": "OMS", "cd": "ecc", "name": "real" }
        ]
    ],
    "object": {
        "kind": "OMV",
        "name": "x"
    }
}
```

Bindings - OMB (1)

- ```
  {
      "kind": "OMBIND",
      /** the base for the cd, optional */
      "cdbase": uri,
      /** the binder being used */
      "binder": omel,
      /** the variables being bound, non-empty */
      "variables": (OMV | attvar)[],
      /** the object that is being bound */
      "object": omel
  }
  ```

- 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`)

Bindings - OMB (2)

- e.g. $\lambda x.\sin(x)$

```
{
    "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)

- ```
  {
      "kind": "OME",
      /** the error that has occured */
      "error": OMS,
      /** arguments to the error, optional */
      "arguments"?: (omel|OMFOREIGN)[]
  }
  ```

Errors - OME (2)

- e.g. to annotate a *division by zero* error in $x/0$

```
{
    "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`

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

- e.g. the term $f(f(\overbrace{f(a,a)}^{y},y),x)$ where the y and x braces mark subterms

References - OMR (2)

```
{
    "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" }]
            }, { "kind": "OMR", "href": "#y" }]
        }, {
            "kind": "OMR", "href": "#x"
        }]
    }
}
```

Summary

- we established that an OpenMath JSON encoding makes using OM much easier in many languages
    - most languages have strutured data types built in
    - serialization into/from JSON exists natively in many languages
    - easy to make use of *Protocol Buffers* or *ZeroMQ* based on this work
- existing approaches had disadvantages, so we developed our own
    - simple to translate to/from the XML Encoding (we have built a translator)
    - uses JSON-native data types
- Thank you for listening. Questions, Comments, Concerns?