Pandoc – JSON Representation

Sébastien Boisgérault, Mines ParisTech, under CC-BY-4.0

February 14, 2016

Pandoc provides a very handy document model; it can be used systematically to generate, analyze and transform pandoc documents. However, it is available as a collection of Haskell types and maybe, *you don't want to learn Haskell!*

You know what? This is fine. Even if pandoc is a virus that spreads Haskell according to its author, you have other options. The standard advice (see "Scripting with pandoc") is to rely on JSON import/export features and then to use your preferred dynamic language – Python, Javascript, etc. – to perform your task.

This article documents the pandoc JSON conversion rules. Read it if you intend to work directly with the JSON data produced by pandoc or to better understand the inner workings and limitations of the pandoc tools that are not based on Haskell, such as pandocfilters.

Getting Started

If you have not done so already, it's probably a good idea to have a look at "Pandoc – Document Structure" for an introduction to the pandoc document model. At the very least, install GHC & the pandoc library, start GHCi and import the following modules:

```
> import Data.Map (Map, fromList, empty)
> import Text.Pandoc.Definition
```

Now, we consider specifically the representation of data as JSON strings; pandoc doesn't reinvent the wheel but instead leverages the aeson package. The function `js` below displays the JSON representation of pandoc data:

```
> import Data.Aeson (encode, ToJSON)
> import qualified Data.ByteString.Lazy.Char8 as BS
> let js x = (BS.putStrLn . encode) x
```

The Rules

The set of rules used to convert pandoc document data to JSON are described in this section. A few exceptions to these rules are the topic of the next one.

Primitive Types

Let's start with the basic types used in the pandoc document model: booleans, integers, doubles and strings:

```
> js True
true
> js 42
42
> js 3.14
3.14
> js "text"
"text"
```

Nothing too surprising so far.

Standard Containers

Consider the standard containers used in Haskell: lists, tuples and maps.

```
> js ["a", "b", "c"]
["a","b","c"]
> js (True, 1, "text")
[true,42,"text"]
> let map = fromList [("a", 1), ("b", 2)]
> js map
{"a":1,"b":2}
```

To summarize:

- Haskell lists and tuples are converted to javascript lists – there is no tuple type in javascript – and maps to javascript objects.
- These conversion rules are applied recursively to container items.

So we have for example:

```
> js ("a", ["b"], [("c", "d")])
["a",["b"],[["c","d"]]]
```

Pandoc Types

The value of a pandoc type is converted to a javascript object with:

- a *type (constructor)* property – key `t` – for the constructor name,
- a *contents* property – key `c` – for the list of (converted) arguments.

Consider for example `MathType`; both its constructors take no argument:

```
> :info MathType
data MathType = DisplayMath | InlineMath
```

Hence, the value `InlineMath` gets converted as:

```
> js InlineMath
{"t":"InlineMath","c":[]}
```

Now, the `Math` (`Inline`) constructor has math type and string arguments:

```
> :info Math
data Inline = ... | Math MathType String | ...
```

Therefore, it is converted as:

```
> js (Math InlineMath "a=1")
{"t":"Math","c":[{"t":"InlineMath","c":[]},"a=1"]}
```

Pandoc Records

Records are converted to javascript objects with:

- a *type* property – key `t` – for the constructor name,
- a property for each record field (the value is converted).

For example:

```
> :info Citation
data Citation
  = Citation {citationId :: String,
              citationPrefix :: [Inline],
              citationSuffix :: [Inline],
              citationMode :: CitationMode,
              citationNoteNum :: Int,
              citationHash :: Int}
```

Hence, we should have:

```
> js (Citation "" [] [] NormalCitation 0 0)
{"t":"Citation",
 "citationSuffix":[],
 "citationNoteNum":0,
 "citationMode":{"t":"NormalCitation","c":[]},
 "citationPrefix":[],
 "citationId":"",
 "citationHash":0}
```

Actually this is not *exactly* what happens because citations fall into the scope of the "single constructor" exception explained in the next section. I would happily show you an example of a pandoc record that is converted according to the general rule ... but the only other one is `Meta` and it's also an exception!

The Exceptions

There are some exceptions to the general rules above. You may also use a less negative wording and call them *optimizations*; indeed, they always reduce the length of JSON representations.

Single Constructor Argument

Consider for example

```
> :info Str
data Inline = Str String | ...
```

Obviously, `Str "text"` should be converted as `{"t":"Str","c":["text"]}`, right ? Think again:

```
> js (Str "text")
{"t":"Str","c":"text"}
```

When constructors have a single argument, the list associated to the `c` key is unwrapped. Be careful: only one list level is removed; for example, for the emphasis

```
> :info Emph
data Inline = ... | Emph [Inline] | ...
```

we have

```
> js (Emph [Str "text"])
{"t":"Emph","c":[{"t":"Str","c":"text"}]}
```

instead of `{"t":"Emph","c":[[{"t":"Str","c":"text"}]]}`.

Single Type Constructor

If a value has a type with a single constructor and you know this type, then the type constructor property of the JSON data is redundant and pandoc gets rid of it.

For example, consider

```
> :info Meta
newtype Meta
  = Meta {unMeta :: Map String MetaValue}
...
```

The JSON representation of the empty metadata `nullMeta` should be `{"t":Meta,"unMeta":{}}`. But if you know that you deal with a value of type `Meta`, given that there is a single constructor (also called `Meta`) for this type, pandoc does not need the type constructor information to interpret the JSON data, and hence, it does not include it. You have instead:

```
> js nullMeta
{"unMeta":{}}
```

Similarly, consider the `Format` type:

```
> :info Format
newtype Format = Format String
...
```

For `Format "html"`, instead of `{"t":"Format","c":"html"}`, we can get rid of the `"t"` key and now, the whole javascript object becomes unnecessary, and only the value of the contents property remains:

```
> js (Format "html")
"html"
```

Does it Work?

Well sure, it works. I mean, pandoc can generate JSON data for a document and read it back unambiguously, so it works. Despite the optimizations performed by the exceptional rules and other sources of ambiguities. It works because at any depth of the interpretation of the JSON data, Haskell has access to the type hierarchy, and therefore knows the type of the data to be evaluated.

But the situation is typically different in a dynamic language, where your code probably knows very little about the pandoc types and expects to read this information at runtime, directly in the JSON data, alongside the values.

If you don't know the type of the data, you face the following ambiguities:

- tuples vs lists (both are JSON lists),
- maps vs pandoc types & records (all of them are JSON objects),
- single constructor argument (list of argument vs single list argument),
- single type constructor (the type information stripped).

The conclusion ? The optimizations – or exceptional rules – used by pandoc in the generation of JSON data are not such a great idea. They may save a few bytes here and there, but they also make JSON much more difficult to use as an exchange format.

What is the solution ? Short of changing the pandoc JSON representation, if we want the same versatility in the processing of documents in Haskell and say in Python, it is to import and use the information about the pandoc document model in Python.