Pandoc – Document Structure

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

February 14, 2016

A specification of the structure of pandoc documents could look like that:

> A document is defined by its metadata and a sequence of blocks.
>
> A block is either a plain text, a paragraph, a code block, ...
>
> A plain text is defined by a sequence of inline elements.
>
> Inline elements are either text, emphasized text, ...
>
> ...

No such document exists, but wait, that may actually be a good thing! What we have instead is a formal definition of the document structure, specified as a collection of Haskell types, documented in Text.Pandoc.Definition on Hackage.

Of course, if you have never used the Haskell programming language, you may have a hard time reading this; I know that because I was in the same place not long ago. Actually, this is not that hard: you only need is an elementary introduction to the Haskell type system, with examples taken from pandoc, and this is what we are doing next.

The Haskell Interactive Environment

To explore the pandoc document structure, we will use the Haskell interactive environment This approach doesn't require pandoc (the command-line tool), but the associated Haskell packages; they may not be available if you have only performed a basic install of pandoc.

Make sure that the Haskell interactive environement GHCi and the Haskell packages associated to pandoc are installed. On Ubuntu for example, you may execute the command

```
$ sudo apt-get install libghc-pandoc.*
```

and then start GHCi with

```
$ ghci
```

In this document, we denote commands executed in GHCi with the `>` prefix; that's a behavior that you can reproduce if you type

```
:set prompt "> "
```

once you have started GHCi.

Pandoc Types

We introduce the constructs of Haskell type system that we need, in the context of the pandoc types (more details about the Haskell type system may be found in the Haskell Wiki).

To make these type availables, load the `Text.Pandoc.Definition` module:

```
> import Text.Pandoc.Definition
```

The Document Type

Pandoc documents are represented as values of type `Pandoc`; to get its definition, type:

```
> :info Pandoc
data Pandoc = Pandoc Meta [Block]
...
```

(the ellipsis denotes details of the output that we have omitted).

Here is how you should read this: the data type `Pandoc` (keyword `data`) is declared with a single constructor, also named `Pandoc`, whose arguments are:

- a value of type `Meta` (that represents the document metadata),
- a list (`[]` notation) of `Block` values (that represent the document itself).

This is actually all the information we need to build a document – or at least the empty document – because `Text.Pandoc.Definition` conveniently provides an empty metadata named `nullMeta`. So we can define

```
> let meta = nullMeta
```

for blocks, use an empty list

```
> let blocks = []
```

and use these values to create an empty document:

```
> let doc = Pandoc meta blocks
```

Blocks and Inlines

Consider the `Block` type:

```
> :info Block
data Block
  = Plain [Inline]
  | Para [Inline]
  | CodeBlock Attr String
  | RawBlock Format String
  | BlockQuote [Block]
  | OrderedList ListAttributes [[Block]]
  | BulletList [[Block]]
  | DefinitionList [([Inline], [[Block]])]
  | Header Int Attr [Inline]
  | HorizontalRule
  | Table [Inline] [Alignment] [Double] [TableCell] [[TableCell]]
  | Div Attr [Block]
  | Null
...
```

Its has 13 different constructors. Some of the constructors have no arguments, for example the horizontal rule

```
> let block = HorizontalRule
```

but most of them have arguments of various types:

- `String`, `Int` and `Double` are Haskell primitive types that represent sequences of characters, integers and (floating-point) numbers,

- The notation `[]` is used for lists – homogeneous sequences of varying length – and the `(,)` notation for tuples – heterogeneous sequences of fixed length. Hence, `[[Block]]` denotes a list of list of blocks and `[([Inline], [[Block]])]` a list of pairs of list of inlines and list of list of blocks (ouch!).

- The burden of complex types based on lists and tuples may be reduced by the use of type aliases (defined with the keyword `type`). The constructs `Attr`, `Format`, `ListAttributes`, and `TableCell` are types aliases; for example:

```
> :info Attr
type Attr = (String, [String], [(String, String)])
> let attr = ("", [], [])
```

- `Format` is declared with the `newtype` keyword

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

  You can think of it as an mere optimization of the `data` construct when the type has a simple constructor whose name is the type name. Otherwise, this type is similar to a classic data type; for example, create a `Format` value with:

```
> let format = Format "html"
```

- The other types used as arguments are pandoc data types; for example

```
> :info Inline
data Inline
  = Str String
  | Emph [Inline]
  | Strong [Inline]
  | Strikeout [Inline]
  | Superscript [Inline]
  | Subscript [Inline]
  | SmallCaps [Inline]
  | Quoted QuoteType [Inline]
  | Cite [Citation] [Inline]
  | Code Attr String
  | Space
  | LineBreak
  | Math MathType String
  | RawInline Format String
```

```
  | Link [Inline] Target
  | Image [Inline] Target
  | Note [Block]
  | Span Attr [Inline]
...
```

You should now be able to understand all constructs used in this definition.

Metadata

Let's import some objects related to the Haskell `Map` type – a key-value store – extensively used in document metadata:

```
> import Data.Map (Map, fromList, empty)
```

The `fromList` function creates a map from a list of key-value pairs and `empty` is the name of the empty map:

```
> empty
fromList []
```

Consider the pandoc `Meta` type that represents document metadata:

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

Let's discard the use of `newtype` keyword – we know that it's similar to `data`. `Meta` has a single constructor which is declared as a record – using the `{}` syntax – with a single field named `unMeta`. The type of this field is a map with `String` keys and `MetaValue` values.

To define values associated to record constructors, we may use named arguments instead of positional arguments; here:

```
> let meta = Meta {unMeta = empty}
> meta
Meta {unMeta = fromList []}
```

The only other use of the record construct among pandoc types is `Citation`:

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

Appendix – Pandoc Types Definition

To display the complete list of pandoc types definition, type in GHCi:

```
> import Data.Map (Map)
> import Text.Pandoc.Definition
> :browse
```

The output of this command sequence, slightly reformated to enhance readability:

```
data Alignment
  = AlignLeft
  | AlignRight
  | AlignCenter
  | AlignDefault

type Attr = (String, [String], [(String, String)])

data Block
  = Plain [Inline]
  | Para [Inline]
  | CodeBlock Attr String
  | RawBlock Format String
  | BlockQuote [Block]
  | OrderedList ListAttributes [[Block]]
  | BulletList [[Block]]
  | DefinitionList [([Inline], [[Block]])]
  | Header Int Attr [Inline]
  | HorizontalRule
  | Table [Inline] [Alignment] [Double] [TableCell] [[TableCell]]
  | Div Attr [Block]
  | Null
```

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

data CitationMode
  = AuthorInText
  | SuppressAuthor
  | NormalCitation

newtype Format = Format String

data Inline
  = Str String
  | Emph [Inline]
  | Strong [Inline]
  | Strikeout [Inline]
  | Superscript [Inline]
  | Subscript [Inline]
  | SmallCaps [Inline]
  | Quoted QuoteType [Inline]
  | Cite [Citation] [Inline]
  | Code Attr String
  | Space
  | SoftBreak
  | LineBreak
  | Math MathType String
  | RawInline Format String
  | Link Attr [Inline] Target
  | Image Attr [Inline] Target
  | Note [Block]
  | Span Attr [Inline]

type ListAttributes = (Int, ListNumberStyle, ListNumberDelim)

data ListNumberDelim
  = DefaultDelim
  | Period
  | OneParen
  | TwoParens

data ListNumberStyle
  = DefaultStyle
```

```
  | Example
  | Decimal
  | LowerRoman
  | UpperRoman
  | LowerAlpha
  | UpperAlpha

data MathType
  = DisplayMath
  | InlineMath

newtype Meta = Meta {unMeta :: Map String MetaValue}

data MetaValue
  = MetaMap (Map String MetaValue)
  | MetaList [MetaValue]
  | MetaBool Bool
  | MetaString String
  | MetaInlines [Inline]
  | MetaBlocks [Block]

data Pandoc = Pandoc Meta [Block]

data QuoteType
  = SingleQuote
  | DoubleQuote

type TableCell = [Block]

type Target = (String, String)
```

Function declarations have been omitted.