An Excursion through Pandoc Types

Massimiliano Farinella

An excursion through the types that constitute Pandoc's abstract syntax tree.

This is an excursion through *Pandoc types*, the elements that constitute the Pandoc abstract syntax tree (AST), the internal document model used by Pandoc to convert any text format it supports.

At the time of this writing, the latest version of *Pandoc types*[1] is 1.23.

1. An example to get started

Every time you convert a document, Pandoc creates an internal representation of it, before writing it again in a different format.

If you want to know about its internal representation, you can export a document in the `native` or in the `json` format. The `native` format is more readable and so it's a good way to learn about the way Pandoc translates every input format into its internal types.

Consider this simple text in Markdown, a title followed by a short paragraph:

```
# A title

A short text.
```

Suppose to save it in a `simple.md` file. Now convert it with:

```
pandoc -f markdown -t native simple.md
```

The result is:

```
[ Header
    1 ( "a-title" , [] , [] ) [ Str "A"
, Space , Str "title" ]
  , Para
    [ Str "A" , Space , Str "short" ,
Space , Str "text." ]
]
```

You can also test it with "Try pandoc!" at *pandoc.org/try/*.

If you treat your text snippet as a full blown, *standalone* document, you can convert it with the `-s` option:

```
pandoc -f markdown -t native -s
simple.md
```

and you get:

```
Pandoc
  Meta { unMeta = fromList [] }
  [ Header
      1 ( "a-title" , [] , [] ) [ Str
"A" , Space , Str "title" ]
  , Para
      [ Str "A" , Space , Str "short" ,
Space , Str "text." ]
  ]
```

It's the same as above, except for the `Pandoc Meta { unMeta = fromList [] }` at the beginning.

[1] The current types' definition can be found at *hackage.haskell.org/package/pandoc-types-1.23/docs/Text-Pandoc-Definition.html*.

2. The structure of a Pandoc document

The general structure of a Pandoc document is:

```
Pandoc Meta [Block]
```

It means that a `Pandoc` document is made by a `Meta` object and a list of `Block` elements. The brackets stand for "a list of" or "an array of".

Since Pandoc is coded in the Haskell programming language, the `native` format follows the syntax of Haskell *constructors*.

Now let's focus on the outer pair of brackets of our example. `Header` and `Para` are two kinds of `Blocks`, so those brackets actually contain a list of (two) `Block` elements.

3. Pandoc Blocks

Pandoc `Blocks` resemble HTML blocks a lot. For those familiar with HTML and CSS, just think of HTML elements that have a default `display: block` CSS property.

In TEX terms, think of elements that live in vertical mode, boxes that stack on top of each other.

Here's their complete list:

- `Plain`
- `Para`
- `LineBlock`
- `CodeBlock`
- `RawBlock`
- `BlockQuote`
- `OrderedList`
- `BulletList`
- `DefinitionList`
- `Header`
- `HorizontalRule`
- `Table`
- `Figure`
- `Div`

Fourteen `Blocks`. Not too many. You can even learn them by heart.

3.1 Headers

The following is the constructor of a `Header`, that represents a heading – i.e. a title – in your document:

```
Header Int Attr [Inline]
```

The first parameter, the one after `Header`, is an integer and it's the level of the heading. Level 1 is the highest one. Even though HTML has `<h1>` to `<h6>`, i.e. 6 levels, I don't think Pandoc puts a limit to the number of levels.

The second parameter is an `Attr`, which is a recurring data structure among Pandoc types, and it's made of three elements:

- an *identifier*; the `id` attribute in HTML, a name that uniquely identifies an element in the whole document, or even in a collection of documents,
- a list of *classes*; the space-separated words of the `class` attribute in HTML; a list of classification labels attached to this textual element,
- a list of *attributes*; they are key-value pairs that constitute a "hash table" – "associative array", "dictionary", choose the name you prefer –; in HTML they would be custom attributes whose names are usually prepended with "`data-`"; it's an additional payload of custom data you can stick to this element.

All the types that have an `Attr` are good candidates to carry additional information in your document. They provide room for customization and arbitrary data that is not in the main text, so not directly interfering with it.

The third parameter is the actual content of the heading: a list of `Inline` elements – remember brackets stand for "a list of" –; this is the "real" content, what goes in the main text.

3.2 Paragraphs

The element for a paragraph is named `Para`; here is its definition:

```
Para [Inline]
```

A paragraph has only one parameter: a list of `Inline` elements.

Suppose you want different kinds of paragraphs in your document, e.g. justified, centered and right aligned ones. You can't, because `Para` has no parameters to diffentiate a `Para` from another one.

Well, it's actually possible, but that kind of extra information is not carried by the `Para` elements.

4. Inlines

`Header` and `Para` are the main elements whose contents are a list of `Inlines`.

If you know a bit of HTML, `Inline` elements are easy to understand. In the TEX world, they'd live in horizontal mode, arranged into lines, usually forming paragraphs.

Here's the complete list, with HTML counterparts, when possible:

- `Str`
- `Emph` (`<em>`)
- `Underline` (`<u>`)
- `Strong` (`<strong>`)
- `Strikeout` (`<s>`)
- `Superscript` (`<sup>`)
- `Subscript` (`<sub>`)
- `SmallCaps`
- `Quoted` (`<q>`)
- `Cite`
- `Code` (`<code>`)
- `Space`
- `SoftBreak`
- `LineBreak` (`<br>`)
- `Math`
- `RawInline`
- `Link` (`<a>`)
- `Image` (`<img>`)
- `Note`
- `Span` (`<span>`)

Twenty `Inlines`. More than the 14 `Blocks`, but still not too many.

4.1 Text and spaces

The `Space` constructor (see the `native` code in the example above) has no parameters and it represents a space or a sequence of spaces, because Pandoc reduces any sequence of spaces to a single space, the same way TeX usually does.

The `Str` constructor is:

```
Str Text
```

where `Text` is a UTF8-encoded string; usually it's a portion of text between two `Spaces`.

I found it may contain spaces too. This should not happen when Pandoc reads a file, but you may create a `Str` with single or multiple spaces in a custom filter or reader, like this:

```
Str "I love    spaces"
```

Pandoc would not complain.

4.2 Styling inlines

A first group of `Inline` elements have the same structure:

```
Emph [Inline]
Underline [Inline]
Strong [Inline]
Strikeout [Inline]
Superscript [Inline]
Subscript [Inline]
SmallCaps [Inline]
```

They are containers of other `Inline` elements and their meaning is straightforward.

A similar one is:

```
Quoted QuoteType [Inline]
```

where `QuoteType` can be either `SingleQuote` or `DoubleQuote`. Their meaning should be apparent too.

4.3 Breaks

The following two `Inlines` have no parameters:

```
SoftBreak
HardBreak
```

The first one is a non-structural line break, like the single newline that counts as a space in TEX, or the newline character in the text of an HTML element.

The second one is structural and forces the following text to start at the beginning of the next line. In HTML, it's a `<br>` element.

4.4 Math from TEX

`Math` shows its TEX lineage:

```
Math MathType Text
```

where `MathType` can be `DisplayMath` or `InlineMath`.

4.5 Citations

`Cite` is for citations. I think BibTEX is among its influences.

```
Cite [Citation] [Inline]
```

It represents a portion of inline text associated to one or more `Citations`. Every `Citation` is an object with six parameters:

```
citationId :: Text
citationPrefix :: [Inline]
citationSuffix :: [Inline]
citationMode :: CitationMode
citationNoteNum :: Int
citationHash :: Int
```

4.6 Inlines with attributes

Then comes the group of `Attr`-carrying `Inlines`:

```
Span Attr [Inline]
Link Attr [Inline] Target
Image Attr [Inline] Target
Code Attr Text
```

`Span` is the most generic `Inline` element, with an `Attr` that may carry extra information.

`Link` and `Image` – think of their HTML counterparts – have also a `Target` parameter, which is a pair of strings: *URL* and *title*.

`Code` is an inline code snippet (a string), with an `Attr` parameter useful to store, for example, the programming language it's written in.

4.7 An Inline made of Blocks

The only `Inline` allowed to contain `Blocks` is

```
Note [Block]
```

which usually represents a footnote.

Pandoc has only one kind of notes, so no footnotes and endnotes in the same document.

Also note that the contents of footnotes is stored in the same place where they are referenced; so, for example, you can't reference the same note more than once in the text, the way you would do with `\footnote[fn1]{...}` and then `\note[fn1]` in ConTEXt.

4.8 Raw material injection

The last `Inline` is

```
RawInline Format Text
```

This is a way to inject custom text, tags or even binary code in your output. The second parameter specifies what is to be injected, but only when the output format is the one specified by the first parameter.

When you make a conversion with an output format that does not match the `Format` parameter, this element is simply discarded, as if it were not there.

This is useful especially in filters, e.g. when you want to add elements that are not natively supported by Pandoc.

5. Back to blocks

We are done with `Inlines`, so let's go back to `Blocks` and let's start with two easy ones:

```
RawBlock Format Text
CodeBlock Attr Text
```

They are the equivalent of `RawInline` and `Code` for `Blocks`.

```
HorizontalRule
```

It's the equivalent of a `<hr>` in HTML and it has no parameters; so all horizontal rules are created equal in Pandoc.

5.1 Blocks of blocks

```
BlockQuote [Block]
```

is the equivalent of `<blockquote>` in HTML and it's a long citation made of blocks. It lacks any further parameter, so – like `Paras` – you can't directly differentiate a `BlockQuote` from another.

The most general-purpose `Block` is `Div`:

```
Div Attr [Block]
```

It's a generic container of blocks, but it has an `Attr` data structure. It's for `Blocks` what `Span` is for `Inlines`.

In case you need a custom textual element that contains blocks and is not currently natively supported by Pandoc, take a `Div` and specialize it with classes and attributes.

`Figure` entered the Pandoc types' family recently:

```
Figure Attr Caption [Block]
```

it's intended for figures, illustrations, etc. and it has an extra parameter, compared to `Div`: `Caption`, whose name tells its intent.

`Caption` is mainly a list of `Blocks`, but it may have an optional shorter version, which is a list of `Inlines`, i.e. a one-liner. Here's the definition:

```
Caption (Maybe ShortCaption) [Block]
ShortCaption [Inline]
```

That weird `(Maybe ShortCaption)` means that the short version is optional. A `Caption` has a list of `Blocks` for sure; it may have a shorter, one line

version too; but it may also not.

5.2 Tables

The most complex `Block` in Pandoc is, by far, the `Table`:

```
Table Attr Caption [ColSpec] TableHead
[TableBody] TableFoot
```

So Pandoc tables have an `Attr`, a caption, and a list of specs of the alignment and width of every column.

Their rows are divided in many sections: a head, many bodies and a foot.

`Caption` is the same as in `Figure`:

```
Caption (Maybe ShortCaption) [Block]
ShortCaption [Inline]
```

A default alignment for every column is specified by `ColSpec`:

```
ColSpec (Alignment, ColWidth)
```

`Alignment` can be only one among

- `AlignLeft`
- `AlignRight`
- `AlignCenter`
- `AlignDefault`

`ColSpecs` can specify column width too: `ColWidth` can be a number in *double* precision more than 0 and less or equal to 1; column widths are relative to the table's width: 0.5 means "half the table width", 1.0 means "full width". `ColWidth` is `ColWidthDefault` when the column width is not explicitly set.

Table head and foot have an `Attr` and are made of rows:

```
TableHead Attr [Row]
TableFoot Attr [Row]
```

You are not compelled to have a head, a foot or multiple bodies – in your tables, I mean.

The simplest table you can fancy has no head nor a foot, and a single body: its `TableHead` and `TableFoot` components will be there anyway, but they will have an empty list of `Rows`, while its list of bodies will have only one `TableBody`.

Table bodies have more parameters than head and foot:

```
TableBody Attr RowHeadColumns [Row]
[Row]
```

Every `TableBody` can have its local headers, spanning the first *n* columns and the first *m* rows.

The *n* quantity of header columns is specified with the `RowHeadColumns` parameter (I think it stands for "header columns for each row").

The *m* quantity is not specified as a number; instead, `TableBody` has two lists of `Rows`: the first ones are headers, the second one carry the actual data, except for their first *n* cells (`RowHeadColumns`).

So *m* is actually the number of rows in the first list.

```
Row Attr [Cell]
```

Rows have an `Attr` and are made of cells.

```
Cell Attr Alignment RowSpan ColSpan
[Block]
```

Every cell has an `Attr`, like every head, foot, body, and row. It may also have an alignment of its own, different from the one specified at table level.

`ColSpan` and `RowSpan` are the number of columns

and the number of rows – respectively – that a `Cell` spans over. Their default value is 1.

Cells are made of `Blocks`: they can't directly contain `Inlines`, as you would expect from HTML.

5.3 Pretty weird to be `Plain`

Table cells are the right element to explain the weirdest `Block`:

```
Plain [Inline]
```

`Plain` is a list of `Inlines`, just like a `Para`, but it's not a paragraph.

Here the analogy with HTML comes in handy, because HTML table cells can contain both inlines and blocks. Here's some examples, all legal in HTML:

```
<td id="cell-1">text</td>
<td id="cell-2"><em>emphasized
content</em> in a cell</td>
<td id="cell-3">
  <p>A first paragraph.</p>
  <p>A second paragraph.</p>
</td>
<td id="cell-4"><img src="..." /></td>
```

In Pandoc, a table cell can only contain a list of `Blocks`; in the examples above, only *cell-3* contains a list of blocks – two paragraphs – while *cell-1* and *cell-4* contain a single inline – a `Str` and an `Image` --; *cell-2* in Pandoc would be a list of seven `Inlines`:

```
[
  Emph [ Str "emphasized", Space, Str
"content"],
  Space,
  Str "in",
  Space,
  Str "a",
  Space,
  Str "cell"
]
```

That's where `Plain` comes in, as a `Block`-wrapper of `Inlines`. In the example above, the contents of *cell-1* would be:

```
[ Plain [ Str "text" ] ]
```

that is a list of one `Block`: a `Plain`; and so the table cell is happy to carry a list of `Blocks` – actually, a minimal list of one element; but it's a `Block`, so it's fine.

`Plain` plays the same role in list items, where the HTML analogy stays true as in table cells.

5.4 Lists

Now here's the list of Pandoc lists:

```
BulletList [[Block]]
OrderedList ListAttributes [[Block]]
DefinitionList [([Inline], [[Block]])]
```

If there were `ListItem`, `TermHeader` and `TermData`, they could be written – in a more meaningful way – like this:

```
BulletList [ListItem]
OrderedList ListAttributes
[ListItem]
ListItem [Block]
DefinitionList [(TermHeader,
TermData)]
TermHeader [Inline]
TermData [Block]
```

But please keep in mind that *`ListItem`, `TermHeader` and `TermData` are not real Pandoc types*; I invented them for sake of clarity.

So a `BulletList` is the analogous of a `<ul>` (un-

ordered list) in HTML, and it's a list of items that are actually lists of `Blocks`.

An `OrderedList` is the analogous of a `<ol>` (ordered list) in HTML, and it's like a numbered `BulletList` with three properties carried by the `ListAttributes` parameter:

```
type ListAttributes = (Int,
ListNumberStyle, ListNumberDelim)
```

The first property – an `Int` in Haskell – is the starting number (usually 1).

`ListNumberStyle` can be one of

- `DefaultStyle`
- `Example`
- `Decimal`
- `LowerRoman`
- `UpperRoman`
- `LowerAlpha`
- `UpperAlpha`

`ListNumberDelim` can be

- `DefaultDelim`
- `Period`
- `OneParen`
- `TwoParens`

A `DefinitionList` has its analogy in `<dl>`, `<dt>` and `<dd>`: they are used to describe "description lists" in HTML.

It's a list of pairs whose first part is a list of `Inlines`, like a paragraph, and the second one is a list of `Blocks`.

They would be good to model glossaries or dictionaries; unfortunately, in Pandoc they lack extra parameters to attach some extra data (e.g. a sort key, a database id, etc.).

5.5 LineBlock

`LineBlock` is the last type in this excursion:

```
LineBlock [[Inline]]
```

it's a list of one-liners, not paragraphs, just lines of text.

A HTML counterpart might be `<pre>`.

So we finished *all* the `Block` and `Inline` elements.

6. Metadata

The document's metadata is optional, and it's completely discarded unless you are producing a *standalone* document (with the `-s` option of the `pandoc` command).

Metadata is stored in `Meta`, do you remember the definition of a Pandoc document?

```
Pandoc Meta [Block]
```

This is the constructor of Meta in Haskell:

```
unMeta :: Map Text MetaValue
```

quite obscure, if you're not familiar with Haskell; think of the metadata of a document as a list of properties, each one with a name (`Text`) and a value (`MetaValue`), which can be one of the following:

```
MetaMap (Map Text MetaValue)
MetaList [MetaValue]
MetaBool Bool
MetaString Text
MetaInlines [Inline]
MetaBlocks [Block]
```

Let's start from the third one (`MetaBool`), the easiest one, a boolean value: `True` or `False`. The next one is a string value (`MetaString`). Then a line of rich text (`MetaInlines`) as a list of `Inlines`; then a list of blocks (`MetaBlocks`), like a list of paragraphs, or an entire document (without metadata).

Climbing the complexity ladder, then comes `MetaList`, a list of any of the previous ones, but also – why not, in an escalation of complexity? – a `MetaList`, but also a `MetaMap`, the last `MetaValue`: a key-value pair, where the key is a string and the value – not unexpectedly – a `MetaValue`.

Metadata is usually encoded in YAML syntax; if you're familiar with it, you can type in as much metadata as you want.

7. That's it

I've covered *all* the Pandoc types of version 1.23. They can be reviewed in a short article like this, yet they can embody a lot of documents, at least their structure and contents, and get them translated into many output formats.

8. Not quite…

What follows are personal opinions on Pandoc types, that are inevitably influenced by the way I use this software; so don't take them as a general advice, especially when I talk about possible extensions.

That said, some words about the limitedness of the Pandoc types definition. I found it a feature, not – sorry – a *limit*.

Pandoc has ways to introduce flexibility and add more information, as well as additional, not natively-supported textual elements, through filters, `RawInline` and `RawBlock` elements, and by conventions (more on this below).

Additional information often goes in `Attr` structures, which are not part of the text body and so are simply ignored by output formats that don't know what to do with them, while enriching the documents written in formats that can use them.

8.1 Pandoc to tidy up documents

You can use Pandoc to remove some unwanted clutter from your texts that come from word processors. You can even convert them to the same format, say `docx` or `odt`.

It's a side-effect of the limited number of textual types of Pandoc.

It's especially useful when the texts come from different sources and authors, and you must first make them consistent with a desired schema and then put them together.

8.2 Conventions to do more

Custom styles are a convention that Pandoc uses to support extra styles for output formats like `docx`, `odt` and `icml`.

Refer to the Pandoc user manual for further information; here I'm focusing on the way you can get additional textual elements leaving Pandoc types definition untouched.

When a portion of text needs an inline style – commonly known as "character style" in word processors – you enclose it in a `Span` element with a `custom-style` attribute whose value is a conventional name, e.g. "Red" for a text to be printed in red ink.

If you need custom paragraphs, you enclose them in a `Div` with the same `custom-style` attribute. You must use a `Div`, because it provides an `Attr` structure that `Para`s lack. The up-side of using an enclosing `Div` is that you need only one `Div` for a sequence of paragraphs of the same, non

standard kind.

This is a convention that other formats, not providing custom styles, simply ignore: those `Span` and `Div` elements are transparent to them, they are simply replaced by their contents.

8.3 More on expanding Pandoc types

At first I found the lack of `Para` customizability too limiting, I would have liked an `Attr` for them to specialize paragraphs.

The Pandoc types definition has been updated and expanded over time. While evaluating a new type, I suspect that its expressibility in Markdown is an important factor. And that works as a limiting factor to the expansion of the types family.

Another limiting factor is that new elements, and their interactions with current ones, must be translated into Pandoc's source code; not to mention that a new specification is always likely to be disruptive for existing workflows.

Some elements, like `Para`, look really barebone, while other ones have seen their specs flourish, as in the table model. Citations are pretty articulated too. It's not something planned from the start, but more the result of the developers' contributions and users' demands.

Anyway I would like the Pandoc types definition to stay lean. I'm against adding arbitrary attributes to every Pandoc type – e.g. adding an `Attr` to every element – as someone proposed in the Pandoc mailing list.

Here's where I see room for expansion:

- an `Attr` for `Note` elements, to support different kind of notes and even multiple references to the same note
- indices: when or whether they will be supported, they'll need specific elements
- expandability for the values of `ListNumberStyle`, `ListNumberDelim` and maybe `Alignment`, along with the specification of a fallback; e.g. `LowerGreek->LowerAlpha`, `UpperGreek->UpperAlpha` for lower and upper Greek letters, that would become the already existing `LowerAlpha` and `UpperAlpha` when the Greek ones are not available in an output format

To support the inclusion of other documents, a `Div` following some conventions is probably enough, no new type needed. See `pandoc-include`[2] or `pandoc-include-doc`[3], a project of mine.

Any different workflow may suggest new features and so new types to add to the Pandoc types definition. But Pandoc already provides means for new features without waiting for the inclusion of your desired types in the official definition, something that may never happen.

[2] *pypi.org/project/pandoc-include*
[3] *github.com/massifrg/pandoc-include-doc*