Generic and well-formed Pandoc

J. Alpuim L.T. van Binsbergen J.P. Pizani Flor

Department of Information and Computing Sciences, Utrecht University

November 5, 2012

Table of Contents

What it does

- Pandoc converts between several markup languages.
- 6 input languages including LaTeX, HTML and Markdown.
- More than 15 output languages.

Universiteit Utrecht

How it works

- Recognizes input language by the file extension (there is an option to override it).
- Parses language into a general Pandoc datatype.
- All printers use the general datatype.

How to break it

In HTML:

- Parsing HTML with non-sensical tags (e.g. <bbl>some text</bbl>);
- Parsing HTML with non-sensical nesting (e.g. *head* tag inside a list);
- Mixed input and *just plain wrong* input does not lead to error messages.

We concluded that the parser was too tolerant.

How do we improve it?

Proposed Solution

1. Implement specific datatypes for each input language
2. With a separate parser to provide us with error messages
   Such that correct parsing gives us a well-formed type
3. Use generic programming for transforming these datatypes into the Pandoc datatype;

- Project Goal: Work this out for one input language

Test-case: XHTML

- Defining a datatype corresponding with the XHTML 1.0 Strict language
- Defining a parser (using parsec) to parse into this type
- This gives us errors messages when parsing incorrect input (however the messages are not really helpful yet)

Obtaining Meta Information

Pandoc meta-data datatype has fields for title, authors and date:

```
data Meta = Meta { docTitle :: [Inline]
                 , docAuthors :: [[Inline]]
                 , docDate :: [Inline] }
```

Our generic function for fetching the title:

```
gTitle :: (GTitle a) ⇒ (a → [Inline]) → GenericQ [Inline]
gTitle g = everything (++) (mkQ [] g)
```

Obtaining Meta Information (cont.)

Given these three classes:

```
class (Data a, Typeable a) ⇒ GTitle a where
    gtitle :: a → [Inline]
class (Data a, Typeable a) ⇒ GMeta a where
    gmeta :: a → Meta
class (Data a, Typeable a) ⇒ GPandoc a where
    gpandoc :: a → Pandoc
```

We may do the following with our HTML datatype:

```
instance GTitle Head where
    gtitle (Head l) = ...
instance GMeta HTML where
    gmeta h = Meta (gTitle (gtitle :: Head → [Inline]) h) [] []
instance GPandoc HTML where
    gpandoc h = Pandoc (gmeta h) (gblocks h)
```

Approach #1

Type Classes

The Pandoc datatype is defined in terms of:

- *Blocks*: The way in which the document is structured (e.g. tables, lists, etc) and contain *inlines*.
- *Inlines*: Specific formats for text (e.g. emph, bold, etc.)

Following the same approach to Meta, two type classes were defined:

- GBlocks, with method *gblocks*
- GInlines, with method *ginlines*

Mutual Recursion

- Pandoc has only top level blocks
- XHTML can have blocks inside inlines, which makes it mutually recursive
- The user has to make a decision on how to deal with it

Instances for XHTML

```
instance GBlocks BlockTags where
    gblocks (Paragraph ts) = Para (ginlines ts) : (gblocks ts)
    gblocks (Div ts) = Plain (ginlines ts) : (gblocks ts)
    gblocks (Header (H1 blas)) = Header 1 (ginlines blas) :
                                   (gblocks blas)
    gblocks (BlockText raw) = [Plain [Str raw]]

instance GInlines InlineTags where
    ginlines (Span ias) = ginlines ias
    ginlines (Em str) = [(Emph [Str str])]
    ginlines (InlineText str) = [Str str]
```

Characteristics

- Genericity of transformation interface
  - Completely general!

Universiteit Utrecht

Characteristics

- Genericity of transformation interface
  - Completely general!
- Level of overhead
  - Potentially a lot of boilerplate code.

Characteristics

- Genericity of transformation interface
  - Completely general!
- Level of overhead
  - Potentially a lot of boilerplate code.
- Level of understanding of Pandoc datatype
  - User needs to be Pandoc-aware.

Characteristics

- Genericity of transformation interface
  - Completely general!
- Level of overhead
  - Potentially a lot of boilerplate code.
- Level of understanding of Pandoc datatype
  - User needs to be Pandoc-aware.
- Level of freedom for the user when defining transformation
  - Free implementation, but choice is required.

Approach #2

A generic AST parser

What it is

The transformation is implemented as a parser *with an output of type Pandoc*. This parser is not an ordinary parser, because:

1. It does *not* parse a sequence of characters or tokens, but the AST of a document (as a Haskell datatype).
2. It is generic over its input type, i.e, it does not know the structure of its input.

A generic AST parser

How it works

The parser is structured as a recursive-descent parser:

- Each concept of Pandoc (one of the possible nodes in a Pandoc-typed value) corresponds to a parsing function.
- These parsing functions are organized in a hierarchy, like in a Parsec parser.
- Each parsing function returns its result inside the *Maybe* monad.
- Each generic parsing function can be *specialized* if necessary.

An example of generic parsing function

The gPandoc function, root of the parsing hierarchy, looks like:

```
gPandoc e m b = (combine . flat2 . gmapQ collect) `extQ` e
  where
    collect child = (m child, b child)
    combine (m, b) = if isJust m && isJust b
      then Just $ Pandoc (fromJust m) (fromJust b)
      else Nothing
```

- We try to get all possible substructures, for each child.

An example of generic parsing function

The gPandoc function, root of the parsing hierarchy, looks like:

```
gPandoc e m b = (combine . flat2 . gmapQ collect) `extQ` e
    where
        collect child = (m child, b child)
        combine (m, b) = if isJust m && isJust b
            then Just $ Pandoc (fromJust m) (fromJust b)
            else Nothing
```

- We try to get all possible substructures, for each child.
- This *collect and combine* behavior is widespread.

An example of generic parsing function

The gPandoc function, root of the parsing hierarchy, looks like:

```
gPandoc e m b = (combine . flat2 . gmapQ collect) `extQ` e
    where
        collect child = (m child, b child)
        combine (m, b) = if isJust m && isJust b
            then Just $ Pandoc (fromJust m) (fromJust b)
            else Nothing
```

- We try to get all possible substructures, for each child.
- This *collect and combine* behavior is widespread.
- Specialize behavior by using the *e* parameter.

An example of generic parsing function

The gPandoc function, root of the parsing hierarchy, looks like:

```
gPandoc e m b = (combine . flat2 . gmapQ collect) `extQ` e
    where
        collect child = (m child, b child)
        combine (m, b) = if isJust m && isJust b
            then Just $ Pandoc (fromJust m) (fromJust b)
            else Nothing
```

- We try to get all possible substructures, for each child.
- This *collect and combine* behavior is widespread.
- Specialize behavior by using the *e* parameter.
- The explicitly-passed parameters *m* and *b* “tie the knot”.

How to use the generic functions and specialize them

How to implement the conversion from a new input language to the Pandoc type?

How to use the generic functions and specialize them

How to implement the conversion from a new input language to the Pandoc type?

```
dummy :: () → Maybe a
dummy = const Nothing
```

```
hPandoc = gPandoc dummy hMeta gBlocks
hMeta = gMeta dummy hTitle hAuthors hDate
hTitle = gTitle eHTMLTitle
hAuthors = gAuthors eHTMLAuthors
hDate = gDate eHTMLDate
```

How to use the generic functions and specialize them

How to implement the conversion from a new input language to the Pandoc type?

```
dummy :: () → Maybe a
dummy = const Nothing

hPandoc = gPandoc dummy hMeta gBlocks
hMeta = gMeta dummy hTitle hAuthors hDate
hTitle = gTitle eHTMLTitle
hAuthors = gAuthors eHTMLAuthors
hDate = gDate eHTMLDate
```

- By *partially applying the generic parsing functions*
- Specializing when desired (minimal matching subtrees)
- "Tying the knot elsewhere"

Universiteit Utrecht

Characteristics

- Genericity of transformation interface
  - Very general.

Characteristics

- Genericity of transformation interface
  - Very general.
- Level of overhead
  - Less overhead than #1. The user writes code only for subtrees in which he wants to override the generic behavior.

Characteristics

- Genericity of transformation interface
  - Very general.
- Level of overhead
  - Less overhead than #1. The user writes code only for subtrees in which he wants to override the generic behavior.
- Level of understanding of Pandoc datatype
  - The same as approach #1, complete.

Characteristics

- Genericity of transformation interface
  - Very general.
- Level of overhead
  - Less overhead than #1. The user writes code only for subtrees in which he wants to override the generic behavior.
- Level of understanding of Pandoc datatype
  - The same as approach #1, complete.
- Level of freedom for the user when defining transformation
  - Customization is possible, but **not** needed.