Generic Pandoc

João Alpuim, Liewe van Binsbergen, João Paulo Pizani Flor

Utrecht University
e-mail: {j.p.pizaniflor,l.t.vanbinsbergen,j.m.queirozataideagorretaalpuim}@students.uu.nl

1 Introduction

1.1 Motivation

There is a big variety of markup languages available. All of them serve the same essential purpose, but still use different syntactic constructs. Often, different organizations and publishers require documents to be written in different markup languages, and this leads to a need for transformation between content in all of these languages.

This task, of taking as input several structurally similar languages, and operating over them, seemed to us a perfect fit to apply generic programming techniques.

1.2 Choice of library

Our original goal was to work with several markup languages in a unified way, trying to apply generic programming to enhance parsing and processing for all of them. Therefore, we searched for a Haskell library which could handle as much markup languages as possible.

This led us to choose the Pandoc library. Pandoc is able to convert (currently) from 7 input languages into 16 output languages (besides plain text), while also being reasonably recent and frequently updated.

2 Analysis

Pandoc interprets all of its supported input languages with only one parser (even though there are different modules for each language). It parses all languages to a common `Pandoc` datatype. This datatype has to be general enough to be able to represent features from all input languages.

Due to this generality, we expected some valid constructs from one language to be allowed as part of the input in a language where these constructs are invalid. Pandoc would be therefore unable to detect these inconsistencies and provide sensible error messages.

To confirm this expectations we ran some tests:

Incorrect nesting in HTML The test HTML file (given as input to Pandoc) had a *head* tag inside a numbered list environment, which is not allowed by the specification. Pandoc gave no error messages and in the resulting PDF, the text inside the *head* tag was ignored.

Invalid tag names We tried to give Pandoc an HTML test input file in which non-existent tag names were used (i.e, ”bbl” and ”pandoc”). Again, no error messages were present during the processing, and the output PDF file contained the text between the invalid tags, as if the tags were not present in the input.

Embed Markdown in HTML We embedded a little markdown-formatted text at the beginning of an HTML file. The goal of this test was to determine whether Pandoc tried to ”guess” in any way the format of the file by looking at its contents. In the resulting output, the whole Markdown-formatted preamble (before the opening <html>) was ignored, which led us to conclude that Pandoc uses *only the file extension* in an *rough attempt* at guessing the input format.

Wrong extension We fed Pandoc some TeX input, but gave the file a *.html* extension. Pandoc then consumed the whole input and produced *no error messages whatsoever*, even though no HTML tags were present in the input. The resulting PDF file just contained the whole TeX code, as if it was plain text.

Summarizing the problems faced during the tests: Pandoc has to parse features of all input languages into a single intermediate representation (modeled by the *Pandoc* datatype). This need for a general representation of all possible inputs leads to ambiguity in the parsing, and - consequently - to incorrect behaviour and lack of error messages when parsing some kinds of invalid inputs.

3 Proposed solution

Specific datatypes As the problems mentioned in section 2 are caused by a too general target datatype for the parser, a big part of the solution is to introduce specific datatypes modeling the structure of each of the input languages. We did not introduce an specific datatype for each input language. However the goal of this project is to make the necessary functionalities available in order to create a library that can be extended with any user-specific input language.

Generic solution By looking at the *Pandoc* input languages we clearly see that all of them provide constructs modeling a certain set of *markup concepts*. In fact, these concepts are precisely represented in the `Pandoc` datatype, that captures the markup concepts common to all of the input languages. These include document meta-information such as the title, authors, etc. and content itself such as paragraphs, lists, etc.

Therefore we thought of using generic programming to define generic functions that traverse the datatype and apply a specific function (defined by the user) that converts a specific node of the AST from the user's datatype into a node of the `Pandoc` datatype.

We opted to use Haskell's type-classes to make these functions for each specific datatype available, since one can always declare a new instance for a new datatype. This would provide the end-user an API where he may only declare these instances, explaining how the program can extract the information from the specific datatype.

We chose to use `Scrap Your Boilerplate` mainly due its extensibility but also because of its easiness-of-use and the built-in functionalities to query and transform data.

Implementation We started off by defining the following class and generic function:

```
type Title = [Inline]

class (Data a, Typeable a) => GTitle a where
gtitle :: a -> Title

gTitle :: GenericQ Title
gTitle = everything (++) (mkQ [] gtitle)
```

Where `[Inline]` is the type of the title in the `Pandoc` datatype.

The idea behind this code was to query the datatype with the function defined by the user, in the GTitle class. This would allow one to define the function `gtitle` on a subtree of the AST for the datatype. However, GHC cannot verify the constraints imposed by `gtitle` at compile time, which leads to the following error:

```
Ambiguous type variable 'b0' in the constraints:
(Typeable b0)
```

```
arising from a use of `mkQ' at (...)
(GTitle b0)
arising from a use of `gtitle' at (...)
```

We therefore slightly changed the implementation:

```
gTitle :: (GTitle a) => (a -> Title) -> GenericQ Title
gTitle g = everything (++) (mkQ [] g)
```

By moving the restriction to the function itself, we lost expressiveness. Now it can only be called if `g` is an instance of `GTitle` and we can just extend it with one function. Despite this disadvantages this approach still suits our needs since it still imposes the constraint and we can apply `gTitle` to any function that satisfies it.

Example: HTML We could then mirror *in the datatype itself* restrictions such as element names (tag names in HTML, command names in LaTeX) and the possible sub-elements of a given element. For example, an excerpt of a datatype definition modeling an HTML document could look like:

```
data HTMLDoc = HTMLDoc HTMLOTag HTMLHead HTMLBody
    HTMLCTag
data HTMLOTag  = HTMLOTag
data HTMLCTag = HTMLCTag

data HTMLHead = HTMLHead HTMLHeadOTag HTMLHeadContents
     HTMLHeadCTag

data HTMLHeadContents = HTMLHeadContents [
    HTMLHeadContents_] HTMLTitle [HTMLHeadContents_]

data HTMLHeadContents_ = Meta | Style | HTMLTitle |
    Link | ...

data HTMLTitle = HTMLTitle ...
```

Looking at the definition of *HTMLHeadContents*, for example, it's immediately obvious that the *head* section of an HTML file *must* contain at least one *title* element. Also, the HTMLHeadContents_ type gives an extensive listing of all the allowed children of HTML-Head.

By mirroring the grammar of the input language in a set of mutually-recursive datatype definitions in this way, we can be sure that *any* value with type HTMLDoc is the result of parsing a valid document, according to the standard.

4 Approaches to the Generic Transformation

4.1 Introduction

By creating specific datatypes for every input language we are able to enforce the well-formedness of the input documents. For obtaining actual values of these AST's separate parsers have to be created. These parsers can provide us with error messages about the structure of the input document, which solves one of the observed problems with the current version of Pandoc.

However, we would also be required to define specific writers for *each transformation*. When we have n input languages and m output languages we would require $(m - 1) \times n$ writers, assuming every input language is also an output language and that we will not transform from any language to itself (hence the -1). This is obviously a lot of work and therefore not desired.

Pandoc already provides us with writers for all the possible output languages and all these writers work on the general Pandoc representation of a document. If we are able to transform our specific AST's to this general representation we can reuse the writers. This is where generic programming might come in handy.

Our goal for the rest of the project will be to provide an 'interface' of functions that help the users in defining a transformation from his specific AST to the general representation.

4.2 Characteristics of the Approaches

The 'quality' of transformation interface can be measured by the following characteristics:

1. *The genericity of the transformation interface*. Ideally the interface should be as generic as possible, so it can work for any language and their corresponding ASTs.

2. *The level of overhead*. The transformation interface should be designed in such a way that it allows the user to write his transformation with minimum boilerplate code.

Approaches We have explored two approaches:

1. Type Class instances, using a a class for obtaining 'Blocks' and a a class for obtaining inline elements called 'Inlines'.
2. Node transformation

Before we will examine these approaches we will first look at a test case.

4.3 Test Case: Obtaining Meta Information

5 Approach #1: Type Class Instances

5.1 Examining Queries

We have seen an approach of obtaining information from our datatype in section 4.3 by using SYB queries. Can we use a similar approach to obtain information about blocks and inlines to construct the Pandoc type?

In order to answer this question we first need a way to obtain pandoc-blocks from our AST. Assuming that all the input languages to be used in the future will be some kind of markup language, they will all contain some form of block elements. Note that the approach proposed in this section should work for any input language, any datatype even, but assuming the input languages is of some form of markup language allows for easier discussion.

It should be the case that any specific datatype of such a input language will contain a type with corresponding constructors for such a block element. For example, looking again at HTML we could have a type BlockTag with constructors Div, Paragraph, etc. We can query our AST for any such constructor and return a value with the corresponding constructor from Pandoc. So for the query approach to work, we require at least some knowledge of the Pandoc type from the side of the user. This leads to another characteristic for our transformation interface.

Characteristic

3 *The required level of understanding of Pandoc and/or its datatype*. The more a user has to dive into the code of the original the less appealing it is to add a new input language or datatype.

A query to dive into our AST to find all values of a type that matches our ASTs block elements can be easily defined when we disregard the contents of such a block element. At the highest level, the Pandoc datatype consists of Meta information and a list of blocks. When we want to transform our AST into the Pandoc type, all we have to do is obtain all the block elements that our part of it by querying for them!

The problems begin to show when we do take the contents of such a block element into account. In the Pandoc datatype there is no mutual recursion. A block either contains inline elements or other blocks. An inline element however only contains other inlines. The Note constructor is an exception, however it is only used for end notes of a document, not for the contents itself.

5.2 Mutually Recursive Inputs

A constraint on our input language in introduced. In order to be able to transform into Pandoc directly our input language *cannot* be *Mutually Recursive*.

This does not mean that we can not use mutually recursive languages as our input language, e.g. HTML is mutually recursive and already an input language of Pandoc.

Instead of introducing impossibilities for our user, anxious to introduce a new language to pandoc, the current construction of the Pandoc type introduces a *degree of freedom* for the user. As is, Pandoc might not deal with the mutual recursive structure of an input language as the user expects it. The way Pandoc currently deals with it is just one of the many alternatives.

To show an example of the choices a user can make, consider the following case. The user has an XHTML document with at the top level a block element. This block element contains an inline element which, in its turn, contains a second block element and also a piece of emphasized text (which is an inline element). To transform this document into the Pandoc type there are multiple options available. The resulting Pandoc type could, for example, contain just one top-level block element and since there are no child-level block elements in the Pandoc type, completely ignoring the second block element. A second result could be a Pandoc type where the second child-level block element is moved to the top-level, thereby ignoring the recursive structure of the input. A third option would be to interpret the child-level block element as an inline element, for example by retrieving all children inline elements.

Hopefully it is clear that there are many other interpretations possible, which ever is used can be made available as a choice to the user. This is what forms the cornerstone of the approach explained in this chapter. It has obvious downsides, as we will see shortly. First a fourth characteristic for qualifying our approaches is introduced.

Characteristic

4 *The level of freedom the user has in defining the transformation*. More freedom for the user means a greater chance of the user to reach his expected result.

5.3 The Instances

The first approach for transforming any AST to the Pandoc type that we worked out uses an approach similar to the querying methods we have seen in section 4.3 except that it doesn't use 'Scrap Your Boilerplate'.

The core question is defining a function of type $\alpha \rightarrow [P.Block]$ where $P.Block$ refers to the type Block from Pandoc and α is the users type or any child element of it. We call this function $gblocks$. Because gblocks will have to operate on more then one type we define it as a method within a type class called $GBlocks$. We do the same for the function $ginlines$ within type class $GInlines$. To summarize: the function $gblocks$($ginlines$) obtains from any relevant type within our AST a list of Pandoc blocks(inlines). The $ginlines$ function, and also $gblocks$ function itself, will be used within the definitions of $gblocks$ to obtain the contents of block elements.

5.4 Test Case: HTML Instances

A lot of instances are required, to see why see Appendix A, but most of these instances are trivial and only are only required to transfer the function call onwards to children. The interesting definitions we worked out are as follows:

```haskell
instance GBlocks BlockTags where
    gblocks (Paragraph ts) = P.Para (ginlines ts) : (gblocks ts)
    gblocks (Div ts) = P.Plain (ginlines ts) : (gblocks ts)
    gblocks (Header (H1 blas)) = P.Header 1 (ginlines blas) : (
        gblocks blas)
    gblocks (Header (H2 blas)) = P.Header 2 (ginlines blas) : (
        gblocks blas)
    ...
    gblocks (Header (H6 blas)) = P.Header 6 (ginlines blas) : (
        gblocks blas)
    gblocks (BlockText raw) = [P.Plain [P.Str raw]]

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

Notice that the instance for BlockTags recurses over child elements and therefore the result is not only the block itself, but also all the blocks that could be found from its children.

The instance for InlineTags does not behave like this, so when looking for inlines only the top-level is taken into account.

This is just *one of the many ways* we could have chosen to implement these functions. Another choice available to the user when looking at HTML is the choice of what to do for tags that do not correspond to any part of the Pandoc type such as form elements and buttons. The current implementation of Pandoc just ignores these tags, while we grant the user a choice what to do with these tags.

5.5 Conclusions

Looking back at the characteristics how does this approach ‘score’?

1. The approach is completely general, it can not only be used for any AST but for any datatype, the user is free to give instances for GBlocks and GInlines for any type.
2. In the example of HTML there is a lot of boilerplate code required to give all the instances since the BlockainerTags, InlainerTags and AllTags types need to be traversed. This is caused by the fact that HTML is a mutually recursive type while Pandoc is not. For other languages, e.g. Markdown the overhead will be a lot less because its structure corresponds more directly with the Pandoc type.
3. It is required for the user to know what constructors create values of types Block and Inline. The choice a user makes on how to implement his own type influences the required knowledge of Pandoc required. The user can go in as deep as he likes, but also decide keep the transformation really simple.
4. As mentioned above, there is complete freedom for the user to implement the interface however he pleases. This is the prime advantage of the proposed approach.

We **conclude** that some degree of knowledge of the Pandoc type is required and that, depending on the complexity of the users AST, this approach potentially leads to a lot of boilerplate. What the approach lacks in this respects, it makes up in the level of freedom that is available to the user. If the user is not prepared to look into the Pandoc datatype this level of freedom might be unwanted. Whether we can prevent any requirements on the knowledge of Pandoc for the user, while maintaining levels of freedom, is something that is explored in the next chapter.

6 Appendix

A The HTML datatype

```
data HTMLTag     = HTMLTag
data HTMLEndTag = HTMLEndTag
data HeadTag     = HeadTag
data BodyTag     = BodyTag
  deriving (Show, Typeable, Data)
data HeadEndTag = HeadEndTag
data BodyEndTag = BodyEndTag

data HTML = HTML Head Body
  deriving (Show,Typeable,Data)
data Head = Head HeadContents
  deriving (Show,Typeable,Data)
data Body = Body [BlockTags]
  deriving (Show,Typeable,Data)

data HeadTags      = Title
                   | Base
                   | Style
  deriving (Eq,Show,Typeable,Data)
type HeadContents = [(HeadTags,Contents)]
type Contents      = String

data BlockTags = Div [AllTags]
               | Paragraph [AllTags]
               | Header Header
               | BlockText String
  deriving (Show, Typeable, Data)
data InlineTags = Span [InlainerTags]
                | Em String
                | InlineText String
  deriving (Show, Typeable, Data)
data ContainerTags = Object [BlockainerTags]
                   | Button [BlockainerTags]
  deriving (Show, Typeable, Data)

type AllTags         = Either BlockainerTags InlainerTags
type BlockainerTags = Either BlockTags ContainerTags
type InlainerTags    = Either InlineTags ContainerTags

data Header = H1 [BlockainerTags]
            | H2 [BlockainerTags]
            | H3 [BlockainerTags]
            | H4 [BlockainerTags]
            | H5 [BlockainerTags]
            | H6 [BlockainerTags]
  deriving (Show, Typeable, Data)
```