Draft for *Typed Final Markup Revisited*

JONAS U. BENN

This paper is mostly a case study of the tagless-final style—which is a solution to the *Expression Problem*. We describe how the tagless-final (TF) encoding can help to create truly extensible representations for markup documents. This is not a new finding and was first used in the Haskell project *HSXML*.

We provide a comparison of the tagless-final encoding with the *algebraic data type* encoding—that *Pandoc* is using—and describe the essential implementation techniques that HSXML's implementation is based on, to create a context aware encoding. This *context aware tagless-final encoding* has great potential for creating a representation, that is

- truly extensible—i.e. in the dimension of *variants* and the dimension of *observers*
- provides strong guarantees in regards to the well-formedness of the created abstract syntax trees

ACM Reference Format:

Jonas U. Benn. 2018. Draft for *Typed Final Markup Revisited*. 1, 1 (March 2018), 11 pages.

Author's address: Jonas U. Benn, mail@benn.in.

2018. XXXX-XXXX/2018/3-ART
https://doi.org/

1 INTRODUCTION

1.1 Motivation

In the age of digital documents, an author of content is confronted with the question which document format to choose. Since every document format has its advantages, one might not want to commit to a specific format to soon.

A series of blog posts might turn into a book or at least a pretty typeset *pdf*. An author might want to give the reader the freedom to read their text on differently sized displays—if the reader has ever tried to read a paper in *pdf*-format on an e-book reader, no further motivation might be needed.

Luckily the problem of decoupling the initial document from output seems to be solved by the rise of markup languages such as Markdown and alike. These types of documents can be easily compiled into all sorts of output formats by programs as *Pandoc* [4].

If the reader has no objections to such a publishing system, they might read no further and write away their next *format-agnostic* document. But if they are interested in how they can easily extend the representation of their document and let a type-checker reason about the *well-formedness* of it, they may find the findings gathered in this paper worth while.

1.2 Type-safe extensibility

This paper mostly outlines the ideas of the work on *HSXML: Typed SXML* [2] and the underlying idea of *Finally Tagless Interpreters* [1, 3].
The *tagless-final style* is a solution to the expression problem [6]. It is closely related to the problem at hand, in that it is concerned with the simultaneous extension of syntactic variants and interpretations of them. This can be seen as an extensibility in two dimensions [add schema like in Object-oriented programming versus abstract data types].

1.3 ?

In short a *tagless-final encoded* representation of documents like *HSXML* has in our opinion two major advantages over markup languages such as Pandoc's internal one:

(1) Guarantee the well-formedness of the document by construction

(2) Easy and full extensibility without loosing the guarantees of 1.

While having these two advantages we still do not want to loose perspective and solve to our initial goal:

(1) Writing documents that are format agnostic—i.e. observe our source in different ways

or as described in the Wikipedia-article on *Markup Languages*

> Descriptive markup
>
> Markup is used to label parts of the document rather than to provide specific instructions as to how they should be processed. Well-known examples include LaTeX, HTML, and XML. The objective is to decouple the inherent structure of the document from any particular treatment or rendition of it. Such markup is often described as "semantic".

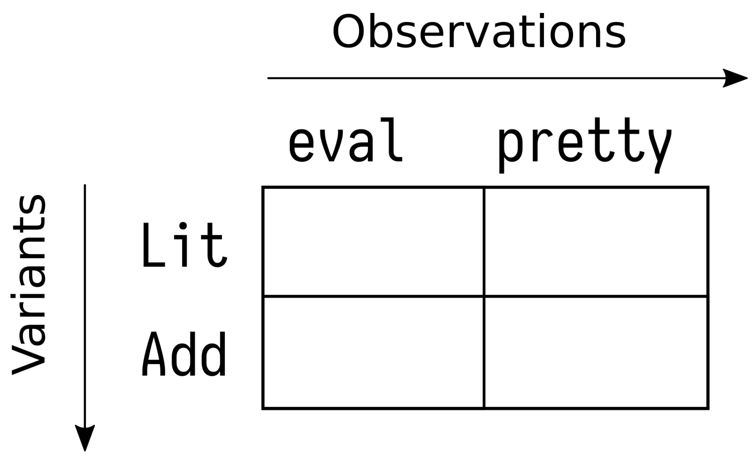

Fig. 1. Dimensions of the *Expression Problem*

2 EXTENSIBILITY OF MARKUP REPRESENTATIONS

2.1 Extensible Observers

Pandoc achieves the separation of input and output format by choosing an algebraic data type (ADT) as its intermediate representation. We will quickly sketch why such an encoding leads to an easy extensibility of constructors by looking a subset of Pandoc's abstract syntax tree (AST) and writing some *observers* for it.

Given the representation (Figure 2) we can write observers that interpret this data in different ways (Figure 3). So in the dimension of observers an ADT encoding is obviously extensible.

Now we can construct a tree in the host language and interpret it in two different ways:

```
groceryList :: [Block]
groceryList
  = [ Heading 1  [ Str "Grocery list"]
    , BulletList [ Paragraph [ Str "1 Banana"]
                 , Paragraph [ Str "2 "
                             , Emph [Str "organic"]
                             , Str " Apples"]]]

groceryListCM :: Markdown
groceryListCM = mconcatMap docToCMark groceryList

groceryListLaTeX :: LaTeX
groceryListLaTeX = mconcatMap docToLaTeX groceryList
```

We can make our life a bit easier by adding an instance for `IsString` for our representation. This injects `String` automatically into our data types by applying `fromString` to it.

```
instance IsString Inline where
  fromString = Str
```

Our initial definition is now even more concise:

```
groceryListShort :: [Block]
groceryListShort
  = [ Heading 1  [ "Grocery list"]
    , BulletList [ Paragraph ["1 Banana"]
                 , Paragraph ["2 ", Emph ["organic"], " Apples"] ]]
```

```
data Block
  = Paragraph   [Inline] -- ^ Paragraph
  | BulletList  [Block]  -- ^ Bullet list (list of items, each a block)
  | Heading Int [Inline] -- ^ Heading - level (int) and text (inlines)

data Inline
  = Str String       -- ^ Text (string)
  | EmDash           -- ^ em dash
  | Emph   [Inline] -- ^ Emphasized text (list of inlines)
  | Strong [Inline] -- ^ Strongly emphasized text (list of inlines)
```

Fig. 2. This is part of Pandoc's ADT-encoded AST modulo EmDash

```
docToCMark :: Block -> Markdown
docToCMark (Paragraph text)     = mconcatMap inlineToCMark text
docToCMark (BulletList docs)    = addLineBreak $ mconcatMap (mappend "-
    " . docToCMark) docs
docToCMark (Heading level text) = addLineBreak $ headingPrefix `mappend`
    mconcatMap inlineToCMark text
 where
  headingPrefix = mconcat $ replicate level "#"

addLineBreak :: Markdown -> Markdown
addLineBreak text = text `mappend` "\n"

inlineToCMark :: Inline -> Markdown
inlineToCMark (Str content)     = fromString content
inlineToCMARK (Emph contents)   = "*" `mappend` mconcatMap inlineToCMark
    contents `mappend` "*"
inlineToCMARK (Strong contents) = "**" `mappend` mconcatMap
    inlineToCMark contents `mappend` "**"
inlineToCMARK EmDash            = "---"

docToLaTex :: Block -> LaTeX
...

inlineToLaTex :: Inline -> LaTeX
...

deleteme$
```

Fig. 3. Observers of ADT encoding

2.2 Extensible Variants

The simple ADT encoding works very well, as long as we have foreseen every variant we might want to create. But as soon as we want to add a new kind of variant—e.g. a node representing the em dash—we are out of luck. Even if we have access to the original ADT-definition and we could add this new variant, this would break all existing observers that were written for the original set of variants.

2.3 Relationship to the Expression Problem

To be extensible in the dimension of observers as well as the dimension of the variants—while still guaranteeing statically their compatibility—is quite a challenge and one that is common when writing software. It was coined as the *Expression Problem* by Wadler [6] and many solutions have been proposed.

The most prominent solutions—that are widely used the Haskell-ecosystem—are described in *Data-types a la carte* [5] and in *Finally Tagless, Partially Evaluated* [1]. Kiselyov's et al. solution to this is, in our opinion, both easy to use and the notation for constructing AST is extremely similar to the ADT-encoded one.

3 SIMPLE TAGLESS-FINAL ENCODING

Our first attempt to encode our document in the tagless-final encoding will not have the distinction between `Doc` and `Inline`—which was enforced by the Pandoc-encoding. But later we will see that we are able to recover that property quite easily with great extensibility properties.

The basic idea of the TF encoding is as follows:

- Create a type class that specifies all our constructors in Church [add footnote with Böhm Berarducci citation] encoding (Figure 4)
- Parametrize over the return-type and recursive fields of those constructors (Figure 5)

The type classes look basically like a GADT-encoding where all recursive occurrences and the return-type are parametrized over.

The observers will now be instances of theses type classes. The reader might notice that we cannot use the same carrier type for different interpretations of our AST—otherwise we would get overlapping instances. This can be quite easily solved by wrapping the carrier type into a *newtype* and add or derive the needed instances for it. In our case `Markdown` is simply a *newtype* of `String`. Therefore the instances for `IsString` and `Monoid` are straightforward to implement.

Figure 6 shows the implementation of an observer in the tagless-final encoding. The implementation is really similar the one in the ADT encoding. But if we have close look, we can see that—since our data type is Church encoded—the observers do not need to be called recursive explicitly. This makes both our code simpler and is essential for extensibility.

```
data Doc = Doc String

instance Monoid (Doc doc) where
  mappend (Doc doc1) (Doc doc2) = Doc $ doc1 `mappend` doc2
  mempty = Doc mempty

-- Constructors

class Block where
  paragraph  ::        [Doc] -> Doc
  bulletList ::        [Doc] -> Doc
  heading    :: Int -> [Doc] -> Doc

class Inline a where
  emDash ::           Doc
  str    :: String -> Doc
  str = Doc
```

Fig. 4. First Step FT-encoding

```
-- DocConstraint defined using ConstraintKinds
type DocConstraint doc = (Monoid doc, IsString doc)

newtype Doc doc = Doc doc

instance DocConstraint doc => -- Have to restrict for the use of 'mempty'
  Monoid (Doc doc) where
  mappend (Doc doc1) (Doc doc2) = Doc $ doc1 `mappend` doc2
  mempty = Doc mempty

-- Constructors

class Block a where
  paragraph  ::        [Doc a] -> Doc a
  bulletList ::        [Doc a] -> Doc a
  heading    :: Int -> [Doc a] -> Doc a

class DocConstraint a =>
  Inline a where
  emDash ::           Doc a
  str    :: String -> Doc a
  str = Doc . fromString
```

Fig. 5. Second Step FT-encoding

```
-- Implement Markdown observer

instance Block Markdown where
  paragraph = fromInline . mconcat
  bulletList = addLineBreak . mconcat . map (mappend (fromInline "\n- "))
  heading level = addLineBreak . fromInline . mappend (mconcat $
    replicate level "#") . mconcat

addLineBreak :: DocAtts doc => DocWithCtx ctx doc -> DocWithCtx ctx doc
addLineBreak (DocWithCtx doc) = DocWithCtx $ doc `mappend` "\n"

instance Inline Markdown where
  emDash = "---"

instance Styles Markdown where
  emph   texts = "*"  `mappend` mconcat texts `mappend` "*"
  strong texts = "**" `mappend` mconcat texts `mappend` "**"


-- Implement LaTeX observer

instance Block LaTeX where
...

instance Inline LaTeX where
...

instance Styles LaTeX where
...
```

Fig. 6. Observer implementation in the tagless-final encoding

Let's see how our example from above looks in our new encoding:

```
groceryList
  = [ heading 1  [str "Grocery list"]
    , bulletList [ paragraph [ str "1 Banana"]]
                 , paragraph [ str "2 "
                             , emph [str "fresh"]
                             , str " Apples"] ]
```

As before, we can automate the injection of `String` into our encoding by using the `OverloadedStrings` language pragma. We do this be adding a constraint on the type classes, so every output format (i.e. carrier type) must have an `IsString` instance.

Interestingly `Doc` has now no dependency on `Inline` anymore and we are now allowed to create the following AST:

```
badHeading = [ heading 1  [ heading 2 [str "Headingception!!"] ] ]
```

As noted above, we lost the distinction between `Doc` and `Inline`. But we also gained something—*Doc* can now be used without *Inline* and we can now also add new variants without changing our original variant definitions:

```
class Styles doc where
  emph   :: [DocWithCtx InlineCtx doc] -> DocWithCtx InlineCtx doc
  strong :: [DocWithCtx InlineCtx doc] -> DocWithCtx InlineCtx doc
```

Not only can we now mix those node types at will, but the type of an expression will reflect which type classes (i.e. algebras) we used for constructing it:

```
stylishNote :: (Inline a, Styles a) => a
stylishNote = strong [str "Green Tea keeps me awake"]
```

That is why the type system can now statically tell us whether we can evaluate `stylishNote` to a particular type.

If we wanted to evaluate an expression, that uses constructors that belong to a type class X and we would want to evaluate the expression to some carrier type C, C has to be instance of X. Since this is a static property, it can be decided at compile time.

3.1 A short note on GHC's Type Inference

When we define an AST like `stylishNote` GHC's type inference might come in our way. If no type signature for `stylishNote` is supplied GHC will try to infer a concrete type for this definition and not the most generalized type.

We can avoid this by either supplying the generalized type signature—as done above—or using the language pragma *NoMonomorphismRestriction*.

4 RECOVER CONTEXT AWARENESS

To regain the context awareness of the Pandoc encoding, we add another field named `ctx` to our `Doc` wrapper (Figure 7). `ctx` is a phantom type and with its help we can specify in which context a constructor can be used. Since phantom types are not materialized on the value level, we are simply using empty data declarations as context types.

```haskell
newtype DocWithCtx ctx doc = DocWithCtx doc
```

Fig. 7. Context-aware wrapper

```haskell
-- Context definitions
data InlineCtx
data BlockCtx
```

As shown before, the first tagless-final encoding had the disadvantage, that we could construct a heading inside another heading. To prohibit this, the heading constructor has the following context-aware definition:

```haskell
class Block doc where
  heading :: Int -> [DocWithCtx InlineCtx doc] -> DocWithCtx BlockCtx doc
  ...
```

The type signature states, that the function expects a `DocWithCtx`-wrapper in the `InlineCtx`-context and returns a wrapper in the `BlockCtx`-context. With this refined signature a heading inside a heading will be rejected by the type system.

To convince Haskell's type system that a conversion from `InlineCtx` to `BlockCtx` is possible, we can use the following type class:

```haskell
class FromInline ctx where
  fromInline :: DocWithCtx InlineCtx doc -> DocWithCtx ctx doc
  fromInline (DocWithCtx doc) = DocWithCtx doc

instance FromInline BlockCtx
```

The set of available contexts should be defined generously, since all independent extensions of the AST should agree on them. This is obviously are restriction—but one that is intended.

It is also possible to create context independent constructors. This can be achieved by parametrizing over the context:

```haskell
class Math doc where
  qed :: DocWithCtx ctx doc
```

5 CONCLUSION

We presented two different encodings that we can choose from for representing a markup language. While the ADT encoding might look like the tool for the job, we have seen that it has some serious limitations. If we do not want to commit to a set of variants forever and do not want to break other people's observers by changing the ADT definition—the tagless-final approach might be a good solution also for this instance of the *Expression Problem.*

For those who want to study this approach in more depth—the lecture notes on *Typed Tagless Final Interpreters* [3] are a great resource.

REFERENCES

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally Tagless, Partially Evaluated”. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 222–238. DOI: 10.1007/978-3-540-76637-7_15. URL: https://doi.org/10.1007/978-3-540-76637-7_15.

[2] Oleg Kiselyov. *HSXML: Typed SXML*. 2014.

[3] Oleg Kiselyov. “Typed Tagless Final Interpreters”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 130–174. DOI: 10.1007/978-3-642-32202-0_3. URL: https://doi.org/10.1007/978-3-642-32202-0_3.

[4] John MacFarlane. “Pandoc: a universal document converter”. In: *URL: http://pandoc.org* (2018).

[5] Wouter Swierstra. “Data types à la carte”. In: *Journal of functional programming* 18.4 (2008), pp. 423–436.

[6] Philip Wadler. “The expression problem”. In: *Java-genericity mailing list* (1998).