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 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 *constructors* and the dimension of *observations*.
- provides guarantees in regards to the well-formedness of the created abstract syntax. These guarantees is ensured by the type system of the host language.

ACM Reference Format:

Jonas U. Benn. 2018. Draft for *Typed Final Markup Revisited*. 1, 1 (March 2018), 15 pages.

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

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

Blank

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 also might want to give the reader the freedom to read their text on different digital devices—e.g. mobile phones, tablets and e-readers.

Luckily the problem of decoupling the initial document from output seems to be solved by the rise of markup languages such as Markdown and the like. These types of documents can be easily compiled into all sorts of output formats by programs such as *Pandoc* [6].

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 syntax 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* [4] and its underlying approach of *tagless-final style* [2, 5].
The *tagless-final style* is a solution to the expression problem [8]. 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—we call those *observations*. This can be seen as an extensibility in two dimensions. For those unfamiliar with the the expression problem—section 2 provides an introduction.

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".

2 BACKGROUND: THE EXPRESSION PROBLEM

The following description of the expression problem is short and precise and stems from Zenger's and Odersky's paper *Indepently Extensible Solutions to the Expression Problem* [9].

> Since software evolves over time, it is essential for software systems to be extensible. But the development of extensible software poses many design and implementation problems, especially if extensions cannot be anticipated. The *expression problem* is probably the most fundamental one among these problems. It arises when recursively defined datatypes and operations on these types have to be extended simultaneously.

In this paper we call those "datatypes" "data variants" or in short "variants" and the operations on them are called "observations". This is inspired by *Extensibility for the Masses* [7].

To get a better intuition on what the expression problem is really concerned with, we will introduce a small extensibility problem and explain the meaning of these *mystical dimensions* with its help:

The task is to find an easily extensible representation of *algebraic expressions*—like e.g. $2 + 4 - 3$—in Haskell. We will present three different encodings and discuss what kinds of extensibility they allow.

2.1 ADT encoding

When encoding algebraic expressions in *algebraic data type (ADT) encoding*, we could write this definition:

```
data Expr
  = Lit Int
  | Add Expr Expr
```

With the above code we defined two data variants, `Lit` and `Add`. Now we can write multiple observations on those variants easily:

```
eval :: Expr -> Int
eval (Lit i)   = i
eval (Add l r) = eval l + eval r

pretty :: Expr -> String
pretty (Lit i)   = show i
pretty (Add l r) =
     "(" ++ pretty l ++ ")"
  ++ "+"
  ++ "(" ++ pretty r ++ ")"
```

We assess that the ADT encoding is extensible in the dimension of observations.

If we wanted to add another variant—e.g. one for negation—we would have to change not only the ADT definition but also all observations. This might be feasible as long as we feel comfortable with changing the original code. But as soon as someone else wrote observations depending on the original set of variants, we risk breaking compatibility.

2.2 OO encoding

If we wanted to ensure that our representation is extensible in the dimension of data variants, we could choose the *object oriented (OO) encoding*. This encoding is centered around the record type `ExprOO` with the following definition:

```
data ExprOO = ExprOO { evalThis   :: Int
                     , prettyThis :: String}
```

This definition states: an `Expr` is a value that can be evaluated to both an `Int` and a `String`. Based on this we can create a set of constructors.

```
newLit :: Int -> ExprOO
newLit i = ExprOO i (show i)

newAdd :: ExprOO -> ExprOO -> ExprOO
newAdd l r = ExprOO evalResult prettyResult
 where
  evalResult   = evalThis l + evalThis r
  prettyResult =
       "(" ++ prettyThis l ++ ")"
    ++ "+"
    ++ "(" ++ prettyThis r ++ ")"
```

This set of data variants can now be easily extended. In the following we define a constructor representing the algebraic operation of *negation*.

```
newNeg :: ExprOO -> ExprOO
newNeg e = ExprOO (- evalThis e) ("- " ++ prettyThis e)
```

These constructors can be used like the constructors of the ADT encoding to create ASTs.

```
exOO :: ExprOO
exOO = newAdd (newLit 4) (newNeg (newLit 2))
```

```
> evalThis exOO
2
```

Although this representation is extensible in the dimension of data variants, the set of observations is fixed by the definition of the `ExprOO` type.

2.3 Church/Böhm-Berarducci encoding

Finally we will choose *Böhm-Berarducci (BB) encoding* for our representation which is the foundation of the tagless-final style.

Böhm and Berarducci used a technique, that is similar to Church encoding, to show that ADTs can be represented by using solely using function application and abstraction in *System F* (i.e. polymorphic lambda-calculus) [1].

In practice this means that we could define lists, instead of using an ADT, the following way:

```
bbList :: (Int -> a -> a) -> a -> a
bbList cons nil = cons 2 (cons 1 nil)
```

To paraphrase the code: if we are supplied one interpretation for the `nil` variant and one interpretation for the `cons` variant—that takes one `Int` and the already evaluated rest of the list—we can interpret the whole list.

We can generalize this idea: To evaluate an AST to a type B, we need to know for each node type, how to evaluate it to B. If all these needed *evaluation strategies* are supplied, we can evaluate (i.e. *fold*) the AST from the leaves up. These *evaluation strategies* are called **algebras**.
This is in essence the idea of Church/Böhm-Berarducci encoding.

In the appendix we include representation that is purely using application and abstraction for our running example. But in Haskell we are luckily not restricted to the features of lambda calculus. Therefore we can use record types for storing *algebras* (i.e. evaluation strategies).

Our algebras are of the following type:

```
data ExprBP a = ExprBP { lit :: Int -> a
                       , add :: (a -> a -> a) }
```

The field called `lit` contains the interpretation for the leaves and therefore is a function that evaluates an `Int` to some type `a`. The `add` function takes two evaluated subtrees as an input—in form of two `a`—and outputs also a value of type `a`.

An AST, using this definition, looks like this:

```
exprBP :: ExprBP a -> a
exprBP (ExprBP lit add) = add (lit 4) (lit 2)
```

This is analogue to the BB encoded list from before, but, instead of getting the algebras for `lit` and `add` one by one, the AST accepts one algebra that bundles both.

2.3.1 *Writing algebras.*

To evaluate those *BB encoded* algebraic expressions, we have to define algebras of the type `ExprBP`:

```
evalExprBP :: ExprBP Int
evalExprBP = ExprBP evalInt evalAdd
  where
    evalInt i   = i
    evalAdd l r = l + r

prettyExprBP :: ExprBP String
prettyExprBP = ExprBP evalInt evalAdd
 where
  evalInt     = show
  evalAdd l r =
       "(" ++ l ++ ")"
    ++ "+"
    ++ "(" ++ r ++ ")"
```

`evalExprBP` can be defined even more concise in Haskell:

```
evalExprBP :: ExprBP Int
evalExprBP = ExprBP id (+)
```

The AST from before, `exprBP`, can now be evaluated by supplying the wanted algebra:

```
> exprBP evalExprBP
6
> exprBP prettyExprBP
"(4)+(2)"
```

We can assess that we could define even more algebras this way and that this encoding is therefore extensible in the dimension of algebras—i.e. observations.

2.3.2 Add variants. We have shown that the BB encoding is extensible in the dimension of observations/algebras but the open question is how to define new variants.

This is can be done by defining a new data type for constructing algebras:

```
data NegBP a = NegBP { neg :: a -> a }
```

Using this data type, we can define two new observations:

```
evalNegBP :: NegBP Int
evalNegBP = NegBP (\e -> -e)

prettyNegBP :: NegBP String
prettyNegBP = NegBP (\e -> "-" ++ e)
```

Finally we are able to use `ExprBP` with `NegBP` in composition:

```
mixedExpr :: ExprBP a -> NegBP a -> a
mixedExpr (ExprBP lit add) (NegBP neg) = add (lit 4) (neg (lit 2))
```

```
> mixedExpr evalExprBP evalNegBP
2
```

Therefore the Böhm-Berarducci encoding is extensible in both dimensions.
If we wanted to add a new data variant, we would need to add a new data type definition for creating corresponding algebras. And we can create new observers by creating values of these data types (i.e. algebras).
As we have shown, these extensions also compose quite well in BB encoding.

Those familiar with the "Scrap your Boilerplate" pattern [3] might notice that the `ExprBP` data type could also be defined as a type class and the algebras—`evalExprBP` and `prettyExprBP`—as instances of this type class. In section 4 we will demonstrate how the encoding with type classes looks like.

3 ADT ENCODED MARKUP

In the last section we tried to find an extensible representation of *algebraic expressions*. Below we will examine how documents written in markup languages like Markdown or LaTeX can be represented.

We will have a look at the representation that Pandoc is using in this section and then show in section 4 how a representation encoded in tagless-final style is an improvement over that.

3.1 Pandoc's internals

Pandoc is a program written in Haskell that parses a document in one format and outputs the content in another format. It can for example create pdf documents out of Markdown files with the help of `pdflatex`.

To translate the content of a document from one format to another, Pandoc first parses the input into an abstract syntax tree (AST). As we have shown in section 2 there a couple of encodings to choose from and Pandoc uses the most common encoding: algebraic data types. In the following we will see what this looks like in Haskell code.

A whole document with its meta data is in Pandoc defined as we would expect:

```
data Pandoc = Pandoc Meta [Block]
```

The list of `Block` represents the content of the document. `Block` is a sum type and has many cases. To get an understanding for this encoding, it is sufficient to consider this subset:

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

The representation is therefore an ADT encoded rose tree. The `Block` type is not only dependent on it self but also on `Inline`. `Inline` represents all elements that can only appear inline in documents. There are also quite many cases of this sum type and this is just a subset of the original definition:

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

The division of the AST cases into the sum types `Block` and `Inline` puts a bit of a restriction on how those ASTs can be constructed. This can be quite helpful when writing parsers and guarantees at least of little bit of well-formedness. For example something like this cannot be constructed due to the differentiation between `Block` and `Inline`:

```
> [ Heading 1  [ Heading 2 [Str "Headingception!!"]]]
<interactive>:25:14:
    Couldn't match expected type 'Inline' with actual type 'Block'
    In the expression: Heading 2 [Str "Headingception!!"]
    In the second argument of 'Heading' namely
      '[Heading 2 [Str "Headingception!!"]]'
```

Given the ADT encoded representation we can write observations that interpret this data in different ways (Figure 1). So in the dimension of observations the ADT encoding is also in Pandoc's case extensible.

We can construct a tree in the host language, Haskell, 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 blockToCMark groceryList

groceryListLaTeX :: LaTeX
groceryListLaTeX = mconcatMap blockToLaTeX 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:

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

3.2 Extensibility Analysis

The simple ADT encoding works very well as long as we have foreseen every constructor we might want to create. But as soon as we want to add a new kind of constructor—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 constructor, this would break all existing observations that were written for the original set of constructors.

As shown in subsection 2.3, we can do better than this using *Böhm-Berarducci encoding*.

```haskell
newtype Markdown = Markdown String deriving (Monoid, IsString)
newtype LaTeX    = LaTeX    String deriving (Monoid, IsString)

blockToCMark :: Block -> Markdown
blockToCMark (Paragraph text)     = mconcatMap inlineToCMark text
blockToCMark (BulletList docs)    = ...
blockToCMark (Heading level text) = ...

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

blockToLaTeX :: Block -> LaTeX
...

inlineToLaTeX :: Inline -> LaTeX
...
```

Fig. 1. Observations of Pandoc's ADT encoding

4 SIMPLE TAGLESS-FINAL ENCODING

In subsection 2.3 we passed the algebras of the *Böhm-Berarducci encoding* explicitly. In the following we will use type classes for the automatic passing of the algebra/observation dictionaries. Using BB encoding with the help of type classes is called *tagless-final style* and was first described by Kiselyov, Carette and Chan in *Finally Tagless, Partially Evaluated* [2].

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 tagless-final encoding is to create a type class that specifies all our variants in Böhm-Berarducci encoding. This is analogue to the *Scrap your type class* variant defined in section 2.3, with the only technical difference:
The observers/algebras are now specified with the help of type classes and the implementations of these specifications are instances of this type class.

4.1 Böhm-Berarducci encoding with type classes

In Figure 2 we can see that the type classes `Block` and `Inline` look similar to the record types from the original BB encoding. Compared to the ADT encoded variants, these are polymorphic in the return type and former recursive fields are now also polymorphic.

These polymorphic parts of the signature consist not only of the type parameter `a`. Instead `a` is wrapped in the *newtype* `Doc`. This has the following reason:
Our first tagless-final encoding could also be realized without this wrapper and `Doc` is not of much use—**yet**. But `Doc` will be vital later on when we add context awareness to our encoding.

4.1.1 Constructing ASTs. In contrast to the encoding in section 2.3, algebras do not need to be passed explicitly to the AST definitions. This leads to even more readable code:

```
groceryList :: (Block a, Inline a) => [a]
groceryList
  = [ heading 1  [str "Grocery list"]
    , bulletList [ paragraph [ str "1 Banana"]]
                 , paragraph [ str "2 "
                             , emph [str "fresh"]
                             , str " Apples"] ]
```

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 and can even be derived using GHC's `GeneralizedNewtypeDeriving` language pragma.

Figure 3 shows the implementation of observers in the tagless-final encoding. The implementation is really similar to the one in the ADT encoding. If we have close look though, we can see one difference: since our data type is Böhm-Berarducci encoded, the observations do not need to be called recursively. This makes both our code simpler and is essential for extensibility.

```haskell
newtype Doc doc = Doc doc

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

-- Algebra specification

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
  str    :: String -> Doc a
  str = Doc . fromString
```

Fig. 2. Böhm-Berarducci encoding using type classes—i.e. tagless-final style

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:

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

4.2 Where tagless-final shines

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 constructors without changing our original constructor definitions:

```haskell
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 we used for constructing it:

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

```haskell
-- Implement Markdown observer

instance Block Markdown where
  paragraph = fromInline . mconcat
  bulletList = ...
  heading level = ...

instance Inline Markdown

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


-- Implement LaTeX observer

instance Block LaTeX where
...

instance Inline LaTeX where
...

instance Styles LaTeX where
...
```

Fig. 3. Observer implementation in the tagless-final encoding

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 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.

4.3 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*.

5 RECOVER CONTEXT AWARENESS

To regain the context awareness of the Pandoc encoding, we add another field named `ctx` to our `Doc` wrapper:

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

`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 simply use empty data declarations as context types.

```
-- 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` data variant has the following context-aware definition:

```
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:

```
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 a restriction—but one that is intended.

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

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

6 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. Especially if our set of data variants might scale up and we would do not want to break other people's observations 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* [5] are a great resource.

REFERENCES

[1] Corrado Böhm and Alessandro Berarducci. "Automatic Synthesis of Typed Lambda-Programs on Term Algebras". In: *Theor. Comput. Sci.* 39 (1985), pp. 135–154. DOI: 10.1016/0304-3975(85)90135-5. URL: https://doi.org/10.1016/0304-3975(85)90135-5.

[2] 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.

[3] Gabriel Gonzalez. *Scrap your type classes*. http://www.haskellforall.com/2012/05/scrap-your-type-classes.html. Blog. 2012.

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

[5] 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.

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

[7] Bruno C. d. S. Oliveira and William R. Cook. "Extensibility for the Masses". In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 2–27. ISBN: 978-3-642-31057-7.

[8] Philip Wadler. "The expression problem". In: *Java-genericity mailing list* (1998).

[9] Matthias Zenger and Martin Odersky. *Independently extensible solutions to the expression problem*. Tech. rep. 2004.