Draft for *HSXML Revisited*

JONAS U. BENN

ACM Reference Format:

Jonas U. Benn. 2018. Draft for *HSXML Revisited*. 1, 1 (March 2018), 7 pages.

1 INTRODUCTION

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`) or 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 initial text from output seems to be solved by the rise of markup languages such as Markdown/Commonmark and alike. These type of documents can be easily [transpiled/converted] into all sorts of output formats by programs as `pandoc`.

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 [bridging the gap between markup and programming languages and/or] how they can let a type-checker reason about the *well-formedness* of their document, they may find the findings gathered in this paper worth while.

This paper mostly outlines the ideas of the work on `HSXML: Typed SXML` and the underlying idea of `Finally Tagless Interpreters`.

In short a richly typed representation like `HSXML` has in our opinion two major advantages over markup languages such as Markdown/Commonmark et al.:

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

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

While having these two advantages we still do not want to loose perspective and be true 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".

Oleg Kiselyov might want to argue that such a markup is even *symantic*.

Author's address: Jonas U. Benn.

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

2 TO BE NAMED

2.1 AST-encoding with Extensible Observers

Pandoc achieves the separation of input and output format by choosing an Algebraic Data Type 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 and writing some observers for it.

Given the representation (Figure 1) we can write *observers* that interpret this data in different ways (Figure 2). So in the dimension of observers our 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 :: CommonMark
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 (integer) 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. 1. This is part of the pandoc AST modulo EmDash

```
docToCMark :: Block -> CommonMark
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 :: CommonMark -> CommonMark
addLineBreak text = text `mappend` "\n"

inlineToCMark :: Inline -> CommonMark
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
...
```

Fig. 2. Pandoc-encoding — Markdown Observer

2.2 Extensible Variants

The 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 node (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 node, this would break all existing observers that were written for the original ADT.

2.3 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 quite common when writing software. It was named as the **Expression Problem** by Wadler [reference] and many solutions have been proposed.

The most prominent solutions — that are right at home in Haskell — are described in *Data-types a la carte* [reference] and in *Finally Tagless …* [reference]. Kiselyov's et al. solution to this is — in our opinion — both easy to use and when used as a DSL for our particular problem, the relationship to S-expressions becomes quite obvious.

2.4 Final Tagless Encoding

Our first attempt to encode our document in the final tagless 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 final tagless encoding is as follows:

- Create a type class that specifies all our constructors in Church encoding (Figure 3)
- Parametrize over the return-type and recursive fields of those constructors (Figure 4)

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.

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

addLineBreak :: DocConstraint doc => doc -> doc
addLineBreak text = text `mappend` "\n"
```

```
data Doc where
  Doc :: String -> Doc

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. 3. 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. 4. Second Step FT-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 must have an `IsString` instance.

Interestingly `Doc` has now no dependency on `Inline` anymore. In a way this is not ideal, since we can now construct the following:

```
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 nodes without changing our original data types:

```
class IsString a => MoreStyles a where
  strong :: [a] -> a
  strikethrough :: [a] -> a
```

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:

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

That is why the type system can now statically tell us whether we can evaluate `stylishText` 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.

2.5 Recover Context Awareness

To regain the context awareness of the Pandoc encoding, we add another field `ctx` to our `Doc` wrapper 5. The `ctx` is a phantom/proxy (?) type and with its help, we can specify in which context a constructor can be used.

As shown before, the first *Final Tagless 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:

```
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.

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 might be very valuable.

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

Fig. 5. Context-aware wrapper

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

```haskell
qed :: DocWithCtx ctx doc
```