Contents

1 Final encoding for extensible markup languages

1.1 Abstract

In the age of digital documents, an author of content is confronted with the question which document format to choose [when it comes to text].

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 [and ongoing popularity) 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 `HSMLX: Typed SXML` and on the underlying idea of `Finally Tagless Interpreters`.

In short a `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 in line with 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* (TODO: add reference).

1.1.1 TODO

Maybe we should mention the the idea of XML is/was exactly that:
(From this website):

> An XML schema provides the following:
>
> - **Validation constraints**: Whenever a typed xml instance is assigned to or modified, SQL Server validates the instance.

- **Data type information**: Schemas provide information about the types of attributes and elements in the xml data type instance. The type information provides more precise operational semantics to the values contained in the instance than is possible with untyped xml. For example, decimal arithmetic operations can be performed on a decimal value, but not on a string value. Because of this, typed XML storage can be made significantly more compact than untyped XML.

1.2 Introduction/Motivation

1. **TODO** Explanation (Explain variants/constructors versus writers/observations)
   In this chapter we will give a quick overview on the challenges related to the representation of markup languages. We will not concern ourselves on how the end-user will construct such in expressions. But to give a short comment on that matter: If we construct our intermediate representation, this could either be constructed directly in the host language as an (shallow/deep?) embedded DSL or we could write some parsers that create our representation in an automated way and hide therefore the implementation details from the user.

1.2.1 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 with the following example:

Given the representation [AST Definition] we can write *writers* that interpret this data in different ways [Markdown & LaTeX/HTML Observer]. So in the dimension of observers our encoding is obviously extensible.

[Example without `IsString` instance]

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

[Code for instance declaration]

The code from above is now much more concise.

```
groceryList :: [Doc]
groceryList
  = [ Heading 1  ["Grocery list"]
```

```
  , BulletList [ Paragraph ["1 Banana"]
               , Paragraph ["2 ", Emph ["fresh"], " Apples"]]]
```

1.2.2 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. inline source code) we have a problem. If we have access to the ADT-definition we could add this new node, but this would break all existing observers that were written for the original ADT.

1.2.3 Expression Problem

To be extensible in the dimension of observers as well as the dimension of the variants and still guarantee statically their compatibility is quite a challenge and one that quite common when writing software. It first was described by Wadler [reference] and many solutions have been proposed.

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

1. **TODO** Tag code and reference

   Code is from `https://github.com/jgm/pandoc-types/blob/master/Text/Pandoc/Definition.hs`

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

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

   type Markdown = String

   docToMd :: Doc -> Markdown
   ```

```
docToMd (Paragraph text) = concatMap inlineToMd text
docToMd (BulletList docs) = concatMap (("- " ++) . docToMd) docs ++ "\n"
docToMd (Heading level text) = headingPrefix ++ concatMap inlineToMd text
  where
    headingPrefix = concat $ replicate level "#"

inlineToMd :: Inline -> Markdown
inlineToMd (Str content) = content
inlineToMd (Emph contents) = "*" ++ concatMap inlineToMd contents ++ "*"

type LaTeX = String

docToLaTex :: Doc -> LaTeX
...

inlineToLaTex :: Inline -> LaTeX
...
```

1.2.4 Final Tagless Encoding

Our first attempt to encode our document in the final tagless encoding will loose the distinction between `Doc` and `Inline` — which was enforced by the Pandoc-encoding. But later we will see that we able to recover that property quite easily [and see how easy it is to enforce these kind of properties with our new encoding].

The basic idea of the final tagless encoding is as follows:

- Create a type class that specifies all our constructors as functions
- Parametrize over the return-type and recursive fields of those constructors

In practice we gain the following encoding of our document format:

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

class Inline a where
  str :: String -> a
  emph :: [a] -> a
```

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

instance Inline Markdown where
  str = fromString
  emph texts = "*" `mappend` mconcat texts `mappend` "*"
```

It basically looks like a GADT-encoding where all recursive occurences and the return-type are parametrized over.

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"] ]
```

[Write something about `NoMonomorphismRestriction`]

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 bad, since we can now construct the following:

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

We lost the distinction between `Doc` and `Inline` — as noted above, but if

1.2.5 Recover Context Awareness

1.3 Overview

In this paper we will start with introducing `HSMLX` — a variant of typed `XML` that uses S-expressions as its syntax — and we will write first constructors and observers for it in Haskell.

1.3.1 SXML

Basically `SXML` is just

1.4 Writing interpreters for typed mark-up

In this chapter we will write two representations for `HSXML`, compare their advantages and disadvantages, and in the end try to come up with an encoding that uses the strengths of both to overcome their shortcomings.

1.4.1 HSXML

Let us first have a look, with what kind of data we are dealing with.

```
<block>
  <h1>Todo List</h1>
  <items>
    <item>Finish term paper</item>
    <item>Reimplemented HSXML</item>
  </items>
</block>
```

```
(block [ h1 "Todo List"
       , items [
```

(Some introduction to `HSXMLS`)

1.4.2 Initial Representation

1.4.3 Final Representation

1.4.4 Ad-hoc polymorphism to the rescue or Final encoding with class

1.5 The decoding problem / No homo!