Expand: Towards an Extensible Pandoc System

Jacco Krijnen[1], Doaitse Swierstra[1], and Marcos O. Viera[2]

[1] Department of Computer Science, Utrecht University, Utrecht, The Netherlands
[2] Instituto de Computación, Universidad de la República, Montevideo, Uruguay
jaccokrijnen@gmail.com, doaitse@swierstra.net, mviera@fing.edu.uy

Abstract. The *Pandoc* program is a versatile tool for converting between document formats. It comes with a great variety of readers, each converting a specific input format into the universal *Pandoc* format, and a great variety of writers, each mapping a document represented in this universal format onto a specific output format.

Unfortunately the intermediate *Pandoc* format is fixed, which implies that a new, unforeseen document element cannot be added. In this paper we propose a more flexible approach, using our collection of Haskell libraries for constructing extensible parsers and attribute grammars. Both the parsing and the unparsing of a specific document can be constructed out of a collection of precompiled descriptions of document elements written in Haskell. This collection can be extended by any user, without having to touch existing code.

The Haskell type system is used to enforce that each component is well defined, and to verify that the composition of a collection components is consistent, i.e. that features needed by a component have been defined by that component or any of the other components. In this way we can get back the flexibility e.g. offered by the packages in the LaTeX package eco-system.

Keywords: Document Formatting, Pandoc, Attribute Grammars, Parsing, Haskell, Type System.

The nice thing about standards is that there are so many to choose from.

— Andy Tanenbaum

1 Introduction

1.1 The Starting Point

The world is littered with document standards, from very simple ones such as *markdown* for easily expressing markup in wiki based systems up to very elaborate ones such as LaTeX, not to mention all the proprietary standards associated with programs like *Word* and its numerous brothers and sisters. It goes without saying that, besides all the differences, these standards have a lot in common, and so do the programs which are used to process and generate documents based on these standards. Unfortunately, once a document is created in one of these formats there is no easy way back; your formatting commands have effectively been stolen by the vendor of your document processing program.

Pandoc is a popular Haskell program which tries to alleviate such problems; its architecture is centered around a "universal document format", together with a collection of *readers* which map documents written using some other format onto this universal format, and a collection of *writers*, each mapping a document represented in this universal format onto the desired output format.

The design of this intermediate format is no sinecure, since on the one hand it is unrealistic to expect that it can represent all document elements which are introduced by any of the existing or future standards, and on the other hand it should not be so restricted that it cannot represents a substantial subset of these elements.

When we look back at the mother of all document formatters, TEX, we see no such limitations, since the language standard contains, besides a collection of primitives, a powerful macro mechanism which can be used to express the formatting of new document elements when the need arises. It is this extensibility which has kept TEX alive and the TEX ecosystem growing over the last 40 years.

A first shortcoming, albeit not such a serious one, is that all formatting commands are following the same lexical and syntactic structure. As a result of this some people prefer to use something like the *markdown* format when typing a document or to use a preprocessor like *lhs2TeX* which was used to add LATEX formatting commands to the input for the paper you are reading, making the Haskell code fragments look good. By using an adaptable syntax there are just fewer symbols to type and the structure of the final document is better visible in the input format of the document.

The second, but probably most serious shortcoming, is that the macro mechanism of TEX can hardly be seen as a modern programming environment. Building abstraction layer on top of abstraction layer by implementing what are effectively programming language interpreters using TEX's macro mechanism, makes resulting systems extremely slow and unforgiving in case the input contains any small mistake. Those who have used *TikZ* in combination with *lhs2TeX* in the *beamer* environment, which may cause a single slide to take seconds to format, can only agree with this observation. Abstraction is nice, but comes at a large cost if the abstraction mechanism itself is expensive. Furthermore the sequential nature of TEX processing makes it cumbersome to collect information and make it available at other places in the output. In those cases we have to recur to writing data into files and reading it back in the next run.

The question we seek to answer in this paper is whether we can deploy an extensible document structure with a way to collect and distribute information in the document, sharing common parts between the various readers and writers, and in which we can describe how an element is to be formatted in a modern, strongly typed programming language.

1.2 Our Approach

In this paper we present our solution to the problems mentioned in the previous subsection, demonstrating the use of the *CoCoCo*[1] libraries written in Haskell, which we developed over the years for constructing compilers in a compositional way [9]. From now on we will talk about *parsers* instead of *readers* and about *semantics* instead of *writers*, thus following conventional compiler construction terminology. The full code can be found in the Haskell package `expand`[2].

One of the libraries we base our solution on is the `murder`[11,10][3] library which can be used to explicitly represent mutually recursive values. In our case these will be grammar fragments which jointly describe the structure of the document to be formatted or converted. Notice that each grammar fragment is represented as a Haskell value, which can be combined, inspected, transformed, abstracted from, etc. Once we have all grammar fragments, which together describe our document, available we can construct the final grammar and map this grammar onto an error correcting parser, using e.g. the `uu-parsinglib`[4] library.

For describing the semantics of the document, i.e. the mapping of the recognised structure onto the desired output format, we use `AspectAG`[13][5]. This library provides a set of combinators for describing attribute grammar based fragments of evaluators. Also here such fragments (or aspects) of the final semantics are described by plain Haskell values, which are to be combined into the overall semantics of the final document structure. Instead of using the fixed *Pandoc* format our parsers and semantics are related to each other by an underlying abstract document format *for this specific class of documents.*

Hence each *Document Element Description* (DED) consists of the following elements:

1. some possibly new document kinds (non-terminals in the grammar) or new element alternatives, thus extending the structure relating the reading and writing phase of the document mapping
2. grammar fragments telling us how to recognise these new elements and how they are to be mapped onto the intermediate document structure
3. common semantics to all possible output formats, such as the construction of a table of contents
4. a description, in attribute grammar terms, describing how to map the newly defined elements onto specific output formats.

1.3 Outline of the Paper

In the paper we will describe how we reimplemented a subset of the intermediate *Pandoc* data type in such a way that it can be easily extended with new document

[1] `http://www.cs.uu.nl/wiki/Center/CoCoCo`
[2] `http://hackage.haskell.org/package/expand`
[3] `http://hackage.haskell.org/package/murder`
[4] `http://hackage.haskell.org/package/uu-parsinglib`
[5] `http://hackage.haskell.org/package/AspectAG`

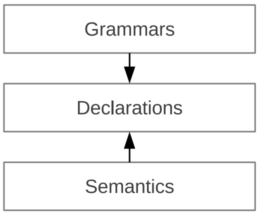

Fig. 1. Architecture

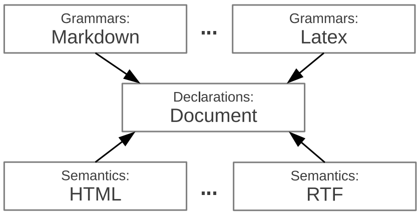

Fig. 2. Multiple Parsers and Semantics

elements. We assume that the reader is familiar with Haskell and its various extensions, since our libraries depend on them. Emphasis will however be on the underlying processes and techniques, and not so much on completeness.

In our example we start by showing how the usual top level structure of a document, including its sections, subsections and paragraphs, may be represented and mapped onto HTML. At the same time we show how some of the micro-formatting, such as bold and italic text are realised. Emphasis will be here on how we express the grammar for the input document, and how to generate some simple output. In no way we claim that something spectacular is going on here; it mainly serves as a basis from which we start to define our extensions in such a way that we can leave the initial code completely intact, and do not even have to recompile it. In the rest of the paper we describe two such extensions: the labeling of section headers with their index number, and the addition of a *table of contents* element, which displays information that is collected from various places in the input text.

2 Implementing `expand`

The architecture of `expand`, which stands for "Extensible Pandoc", is depicted in Figure 1; boxes represent (groups of Haskell) modules and arrows denote **import** relations.

The `expand` library is divided into three parts:

1. **Declarations** of abstract syntax for the general document format
2. **Grammars** that describe the parsers for concrete syntax of input languages.
3. **Semantics** that describe the unparsing for the concrete syntax of output languages.

each of which contain modules that serve as a collection of building blocks for the programmer.

As we show in Figure 2, multiple parsers (e.g. *markdown*, LaTeX) and semantics (e.g. generating HTML, RTF) can be defined following the same approach as *Pandoc* does. What makes the difference between our approach and *Pandoc* however, is that we can also extend, in a modular way, the parsers, semantics and intermediate representations of documents. For example, in Figure 3 we extend the generation of HTML by adding a numbering system to the headers.

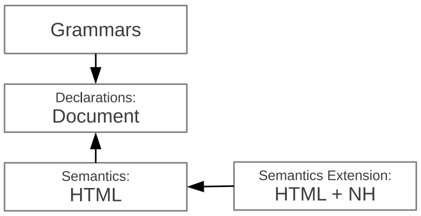

Fig. 3. Generating HTML and Numbered Headers

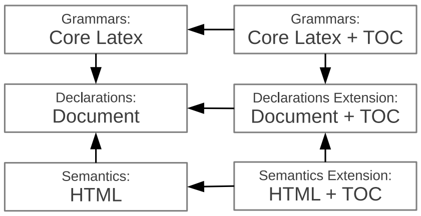

Fig. 4. Documents with Table of Contents

Notice that the original modules are neither inspected nor modified; they are just imported. Thus there is no need to access to the source files of the former semantics, which could have been distributed as binary code. In our approach, extensions can be done to any of the three parts that compose to a complete definition. For example, in Figure 4 we show how extensions to the grammar and the intermediate document type and the semantics (may) depend on earlier modules. Here we start with a subset of LaTeX which does not include the possibility to define a table of contents and to which we will refer to as the LaTeX core; we extend the parser to recognise the `\tableofcontents` command, the document structure to represent its abstract syntax, and the semantics describing how to collect the information, distribute the information in the document, and the final formatting of this table. Note that the first two of these semantic aspects are likely to be defined separately, since they are not HTML specific and can be shared between different output formats.

In the following subsections we will show what such definitions look like. As an example we will show how to construct a program that translates core LaTeX to HTML. We will also show how the extensions of figures 3 and 4 are expressed in Haskell.

2.1 Declarations

In our attribute grammar fragments we use names[6] to refer to children of nodes, names to refer to attributes and names to refer to the non-terminals of the abstract grammar. We use Template Haskell to generate such names from conventional Haskell data type definitions as in Figure 5.

A document (*Document*) consists of a list of blocks (*BlockL*), each being either a header or a paragraph. A header consists of an *Int* representing its level (*level_header*), and its text (*InlineL*). A paragraph contains text, some of which can be bold or italic. The function *deriveAG* generates the necessary labels and types to be used in the attribute grammars fragments describing computations over trees described by the above types. The function *deriveLang* generates a record data type containing a field for each non-terminal. Such a field holds the function which maps the inherited attributes of the corresponding non-terminal to its to synthesised attributes. Once such a record is constructed by combining

[6] We use the `HList` label model as defined in the module *Data.HList.Label4*.

```
data Document = Document {blocks :: BlockL} deriving Show
type BlockL  = [Block]
data Block   = Header    {level_header   :: Int
                         , inlines_header :: InlineL}
             | Paragraph {inlines_par    :: InlineL}
      deriving (Show)
type InlineL = [Inline]
data Inline  = Plain     {str_plainInl   :: String}
             | Bold      {inlines_boldInl :: InlineL}
             | Italics   {inlines_italInl :: InlineL}
   deriving (Show)
 $ (deriveAG ''Document)
 $ (deriveLang "Doc" [''Document, ''BlockL, ''Block, ''InlineL, ''Inline])
```

Fig. 5. The Haskell data types describing our document structure

```
document  → block*
block     → paragraph | header
paragraph → "\begin"  "{" "paragraph" "}" inline* "\end" "{" "paragraph" "}"
header    → "\section"        "{" inline* "}"
          | "\subsection"     "{" inline* "}"
          | "\subsubsection"  "{" inline* "}"
inline    → "\plain"  "{" text    "}"
          | "\textbf" "{" inline* "}"
          | "\textit" "{" inline* "}"
```

Fig. 6. The EBNF for our input language

all attribute grammar fragments for all non-terminals we pass it to the parser, so the parser can apply the appropriate function for each recognised non-terminal. Notice that we use a deforestated approach: the intermediate tree never comes into existence, but is instead directly represented by its semantics, i.e. a function mapping the inherited attributes of the root to its synthesised ones. We use plenty of **type** synonyms, so we have names for all types that play a role as non-terminal in the actual parser.

2.2 Grammars

In this subsection we show how to construct a parser. For a deeper explanation and more information on the types involved, see [9] section 3.3. For simplicity reasons we assume here that plain text is explicitly marked using the commands `\plain{...}` and `\begin{paragraph}`. Such commands can be inserted by a preprocessor, or be omitted by writing a more elaborate parser. The EBNF expressing the concrete input syntax is given in Figure 6, where *text* is a string, excluding the special characters: `\`, `&`, `%`, `$`, `#`, `_`, `{`, `}`, `~` and `^`.

```
gLatex sem = proc () → do
  rec
    document  ← addNT ≺ ⟦ (pDocument sem) blockL ⟧
    blockL    ← addNT ≺ pFoldr (pBlockL_Cons sem, pBlockL_Nil sem)
                                ⟦ block ⟧
    block     ← addNT ≺ ⟦ header ⟧ <|> ⟦ paragraph ⟧
    paragraph ← addNT ≺ ⟦ (pParagraph sem) "\\begin" "{" "paragraph" "}"
                                            inlineL
                                            "\\end" "{" "paragraph" "}" ⟧
    header    ← addNT ≺ let h (x, name) = ⟦ (pHeader sem x) "\\" name
                                                              "{" inlineL "}" ⟧
                            headers     = [(1, "section")
                                          , (2, "subsection")
                                          , (3, "subsubsection")]
                        in  foldr1 (<|>) (map h headers)
    inlineL   ← addNT ≺ pFoldr (pInlineL_Cons sem, pInlineL_Nil sem)
                                ⟦ inline ⟧
    inline    ← addNT ≺ ⟦ (pPlain sem)   "\\plain"   "{"
                                          (someExcept "\\&%$#_{}~^") "}" ⟧
                    <|> ⟦ (pBold  sem)   "\\textbf"  "{" inlineL "}" ⟧
                    <|> ⟦ (pItalics sem) "\\textit"  "{" inlineL "}" ⟧
  exportNTs ≺ exportList document ( export cs_document  document
                                  ∘ export cs_blockL    blockL
                                  ∘ export cs_paragraph paragraph
                                  ∘ export cs_header    header
                                  ∘ export cs_inline    inline
                                  ∘ export cs_inlineL   inlineL)
```

Fig. 7. Our EBNF encoded as a series of grammar transformations

With the abstract and concrete syntax in mind, we use combinators from the `murder` library to straightforwardly encode this grammar fragment in Haskell, as shown in Figure 7. Note that we can freely use Haskell abstractions where this comes in handy, as in the case where we deal with various levels of section headers; as a result our abstract grammar is more expressive than our input grammar.

2.3 Arrows and Their Syntax

A grammar fragment in the `murder` library is expressed using the arrow interface, which generalises the notion of a function, modelling effectful computations with input and output. In our case we maintain a state containing an environment holding the productions for each of the non-terminals introduced thus far.

Because arrow syntax [3] can be a bit daunting, we give some analogies to functions. When writing **proc** *inp* → ... (arrow abstraction), we define the arrow equivalent of writing $\lambda inp \rightarrow$ ... for functions. The *pat* ← *a* ≺ *alternatives* syntax is used in a recursive do block (**do rec**), indicating that we apply the arrow *a* to *alternatives* and match the output to the pattern *pat* (≺ is written as `-<` in Haskell code). Such a **do** block allows for recursive bindings, similar to a **let** block. Finally, we indicate the output of the grammar fragment with *a* ≺ *input* (which should be the last statement in the do block), meaning that the output of arrow *a* will be the final output of our grammar fragment arrow.

In the case of grammar fragments, the input of such an arrow provides information how to refer to earlier introduced elements of the grammar under construction (in this case we call the fragment a *grammar extension*). The empty structure () indicates that our fragment does not need to refer to any other fragment (we say the current fragment is an *initial grammar*).

In Figure 7 we introduce new non-terminals using the *addNT* arrow. A call to *addNT* extends the state with a new non-terminal, it takes the initial productions of this new non-terminal as input and returns a reference to the newly introduced non-terminal, which we can use as non-terminal in further fragments. Fragments as defined in Figure 7 are combined by means of arrow composition, as we will see later.

Each production is expressed using the so called *idiom brackets*[7] (`iI` and `Ii` in Haskell code). The brackets enable a notation which closely follows the common CFG notation, but reduce to normal applicative combinators. We have used class overloading to let the type of each element decide what kind of parser to construct. For example, when we write:

⟦ (*pBold sem*) `"\\textbf"` `"{"` *inlineL* `"}"` ⟧

we construct a parser that parses the strings `"\\textbf"` and `"{"`, next applies the parser for the non-terminal *inlineL* and finally parses the string `"}"`. The strings are not used and the result of the complete parse is constructed by selecting the appropriate semantic function (*pBold*) from the overall semantics *sem*, and applying it to the result of the parser *inlineL*.

Using the function *exportList* a list of non-terminals is constructed that can be used in later extensions. Its first argument expresses that the starting point of the grammar is *document* and that the extensible non-terminals are *document*, *blockL*, *paragraph*, *header*, *inline* and *inlineL*; they can be accessed using the labels *cs_document*, *cs_blockL*, *cs_paragraph*, *cs_header*, *cs_inline* and *cs_inlineL* which were generated by Template Haskell.

Note that the above grammar fragment is parameterised with a record *sem* containing for each production its associated semantic function. The type of this record is imported from the *Declarations* modules and was generated by *deriveLang*. In this way we have decoupled what to do with the recognised structure form the recognition process itself.

[7] `http://www.haskell.org/haskellwiki/Idiom_brackets`

2.4 Semantics

We can compute useful information from an abstract syntax tree by using an attribute grammar. In an attribute grammar, each node in a parse tree is decorated with a set of values, called attributes. There exist two kinds of attributes: synthesised and inherited. Synthesised attributes are used to pass information up to the parent node, while inherited attributes are used to pass information down to children nodes. Attribute value computations can refer to inherited attributes of the parent and synthesised attributes of the children. Attribute grammar based specifications differ from function definitions in the way that in case of the latter we have to specify both all arguments at the same time, and the various parts of a computed result together in the body at the same time, whereas in the former situation this specification can be given incrementally. For a proper understanding it suffices to see each introduction of an inherited attribute as adding an extra parameter to the semantics of a non-terminal and each introduction of a synthesised attribute as an extension of its result.

We will now show how, using the `AspectAG` library, we define a synthesised attribute containing the HMTL code for a piece of parsed input text. This library allows us to define attribute grammar fragments which can be type-checked, compiled, distributed and composed as any normal Haskell value. For naming the individual attributes of a node we follow the same approach as we did with naming the children and the non-terminals: we again use heterogenous lists [5] (`HList` package), in which values of different types can be stored and accessed by using a unique type as index.

Depending on whether we think of the abstract syntax as data type, a tree, or a grammar we use the following words as synonyms:

1. "data type", "parent node" and "left-hand-side (non-terminal)"
2. "data constructor", "current node" and "production"
3. "data constructor field", "child node" and "right-hand-side non-terminal"

Before introducing the definitions of the new attribute, we first create a unique label *html*, using the Template Haskell function *attLabels*:

```
$ (attLabels ["html"])
```

For every production of the abstract syntax with which we associate the synthesized *html* attribute, we provide a *rule* that states how to compute that attribute *html* (a *String*); we use the *syn* function to specify the rules:

```
document_html   = syn html $ do blocks ← at ch_blocks
                                return $ blocks # html
blockLnil_html  = syn html $ return ""
blockLcons_html = syn html $ do block  ← at ch_hd_BlockL_Cons
                                blocks ← at ch_tl_BlockL_Cons
                                return $ block # html ++ blocks # html
```

We define the rule for the only production (constructor) of the *Document* type and the two productions of the *BlockL* type (derived form the list type definitions). We use the *Reader* monad to get access to a small heterogenous record containing the attributes of the child nodes (constructor field). The (#) operator is used to access the fields of those records. Thus, to compute the *html* attribute for the production *Document*, we just return the value of the *html* attribute of its only child. Note that we are not working with the actual data types, but merely use the labels (i.e. *ch_blocks*, *ch_hd_BlockL_Cons* etc) that were *generated* from the data types (this method is key to achieve extensibility of the AST). We show two more rules, for the *Block* productions:

```
header_html    = syn html $ do level ← at ch_level_header
                               inls  ← at ch_inlines_header
                               return $ "<h" ++ show level ++ ">"
                                        ++ inls # html
                                        ++ "</h" ++ show level ++ ">" ++ "\n"
paragraph_html = syn html $ do inls  ← at ch_inlines_par
                               return $ "<p>" ++ inls # html ++ "</p>" ++ "\n"
```

In order to construct the semantic record using the defined attribute rules, we use the generated function *mkDoc* (which explains the role of the "Doc" parameter in the Template Haskell) which was also generated by *deriveLang* (see section 2.1), and which collects the semantic rules for all productions. The definitions of the other functions follows the same pattern as above.

```
semHtml = mkDoc blockLcons_html
                blockLnil_html
                bold_html
                document_html
                header_html
                inlineLcons_html
                inlineLnil_html
                italics_html
                paragraph_html
                plain_html
```

The *mkDoc* function returns exactly the record structure with which we parameterised the grammar fragments.

2.5 Composing the Tool

Now that we have a definition for the semantics, we can finally put the tool together that maps LaTeX onto HTML. We start by writing a small utility function to build the converter:

```
buildConverter gram att input = let parser = compile $ closeGram gram
                                    res = result (parse parser input)
                                in  res emptyRecord # att
```

Thus, *buildConverter* takes an extensible grammar (e.g. a grammar fragment), an attribute with which we can index in the heterogenous record with the synthesized attributes of the root element, and an input string for the parser. We use the `murder` functions *compile* and *closeGram* to generate the parser, and *parse* to run it. With *result* we drop extra information from the parsing process and obtain the result: a function that takes a heterogenous record with inherited attributes (in our case none, thus *emptyRecord*) and returns a record containing the synthesized attributes, of which we select the one specified using (#).

We can now construct a converter tool for our language, by passing the defined semantic functions to the parser description, and using *buildConverter* to build and run the parser.

```
latex2html :: String → String
latex2html = buildConverter (gLatex semHtml) html
```

3 Extending Our Definitions

In this section we show how our design can be extended in three different ways: extending the set of attributes, adding new non-terminals to the abstract syntax and extending the grammar describing the input language.

3.1 Extending the Semantics: Numbered Headers

As a first use case, we extend the HTML generation. In LaTeX section headers are automatically numbered. In order to integrate this aspect into the HTML generation, we define some extra attributes.

We model a header number as a value of type [*Int*], taking the level of headers into account, e.g. 3.1.4 is represented as [3,1,4]. We write a small function to format such an index

```
formatNH :: [Int] → String
formatNH = intercalate "." ∘ map show
```

We introduce an attribute *cHeaderNum*, a chained header number, which threads (or chains) the header indexes through the tree, updating it whenever a header is encountered. Such a chained attribute is by convention a pair of an inherited and a synthesised attribute having the same name, and has the same effect as using a *StateT* monad transformer. We also define a *local* attribute *headerNum*, which is only accessible from within the header node.

```
$ (attLabels ["cHeaderNum", "headerNum"])
  -- the non-terminals which have chained cHeaderNum attributes:
cHeaderNum_NTs = nt_BlockL .*. nt_Block .*. hNil
  -- by default the attribute is copied in a state monad like fashion:
default_cHeaderNum    = chain cHeaderNum cHeaderNum_NTs
  -- initialise the list of Integers at the root of the document:
document_cHeaderNum = inh cHeaderNum cHeaderNum_NTs $ do
                                      return (ch_blocks .=. ([] :: [Int]) .*. emptyRecord)
  -- compute a local attribute containing the new list of numbers:
```

```
header_headerNum     = loc headerNum $ do
                          lhs    ← at lhs
                          level  ← at ch_level_header
                          return $ updateHeaderNum level (lhs # cHeaderNum)
  -- return the updated list of numbers:
header_cHeaderNum    = syn cHeaderNum $ do
                          loc    ← at loc
                          return $ loc # headerNum

  -- auxiliary function which computes the next header number:
updateHeaderNum :: Int → [Int] → [Int]
updateHeaderNum level par = zipWith (+) par' (zeros ++ [1])
  where par'  = par ++ repeat 0
        zeros = replicate (level − 1) 0
```

Note that the computation of this new attribute is independent of the output language. Therefore, this attribute definition is defined in a separate module and can be shared across different output languages. It is now easy to access this attribute in our new definition of the *html* generation, where *synmodM* creates a rule that will overwrite the original rule when extending it.

```
header_html' = synmodM html $ do level ← at ch_level_header
                                 inls  ← at ch_inlines_header
                                 loc   ← at loc
                                 let num = loc # headerNum
                                 return $ "<h" ++ show level ++ ">"
                                          ++ formatNH num ++ " "
                                          ++ inls # html
                                          ++ "</h" ++ show level ++ ">" ++ "\n"
```

We can now construct a new semantic record for *html* generation by combining both the *html* and the *cHeaderNum* aspects:

```
semHtml' = mkDoc (default_cHeaderNum `ext` blockLcons_html)
                 (default_cHeaderNum `ext` blockLnil_html)
                 bold_html
                 (document_cHeaderNum `ext` document_html)
                 (header_headerNum `ext` header_cHeaderNum
                                   `ext` header_html'
                                   `ext` header_html)
                 inlineLcons_html
                 inlineLnil_html
                 italics_html
                 (default_cHeaderNum `ext` paragraph_html)
                 plain_html
```

This is where the actual composition of semantics happens. The original *header_html* rule is extended from right to left, with rules for *cHeaderNum*, *headerNum* and a redefinition for the *html* attribute.

3.2 A Table of Contents

As our second extension we show how to extend the grammar, abstract syntax and semantics by computing a table of contents of the document.

We start out by computing the table of contents as a synthesised attribute, since this computation requires no extension of the abstract syntax. Again, we would like this attribute to be reusable for different output languages, so we model the table as a value of type $[([Int], String)]$, i.e. a list of section headers tupled with the name of the section.

We start by defining two attribute labels: *sToc* contains the synthesised table of contents, and *toc* the complete table of contents, to be passed down the tree as an inherited attribute. In this way information collected from all over the document is made available at all places where we might insert the table of contents.

```
$ (attLabels ["sToc", "toc"])
sToc_NTs = nt_Document .*. nt_Block .*. nt_BlockL .*. HNil
default_sToc = use sToc sToc_NTs (++) []
header_sToc = syn sToc $ do loc ← at loc
                            inls ← at ch_inlines_header
                            return [(loc # headerNum, inls # sInlStr)]
```

For *sToc* we provide a default rule that aggregates the synthesized table of contents. The *use* function from the `AspectAG` library takes an operator to combine the synthesized tables from all the child nodes, and a default value if a child does not define the attribute. Next, we write a specific rule defining how to synthesize a table of contents at the header node. We ask for the local attributes, and reuse the *headerNum* attribute, defined in the previous subsection. We also use the *sInlStr* attribute which formats the *InlineL* text as a simple string, while ignoring text formatting such as bold and italics (we omit its implementation).

We now add an extension to the abstract syntax to be able to indicate where the table of contents is to be inserted:

```
data EXT_Block = Toc
$ (extendAG '' EXT_Block [])
$ (deriveLang "DocToc" ['' EXT_Block])
```

The *EXT_Block* should be read as an extension of the *Block* data type (defined in section 2.1), thus introducing a new production for the table of contents. The function *deriveLang* will also produce a new record type containing the semantic function of the *Toc* production. Now that we have this semantic record available, we can extend the LaTeX grammar

```
gLatexToc sem = proc imported → do
                  let block = getNT cs_block imported
                  toc ← addNT ≺ [[ (pToc sem) "\\tableofcontents" ]]
                  addProds ≺ (block, [[ toc ]])
                  exportNTs ≺ imported
```

We retrieve the non-terminal *block* defined in the fragment from section 2.2 and introduce a new non-terminal *toc* that recognises the LaTeX command. We then add this new non-terminal as an extra alternative to the block non-terminal. Now we can also define the synthesis of the *html* attribute for the *Toc* production:

```
toc_html = syn html $ do lhs ← at lhs
                         return $ formatToc (lhs # toc)
formatToc :: [([Int], String)] → String
formatToc = foldr f ""
  where f (x, section) table = "<a href=#" ++ show x ++ ">"
                              ++ (formatNH x) ++ "  " ++ section
                              ++ "</a><br />\n" ++ table
```

We use the inherited attribute *toc* (the complete table) and format it using a small helper function. From this, we can derive the required semantic record.

```
semHtmlToc = mkDocToc (default_toc ‘ext‘ toc_html ‘ext‘
                       default_cHeaderNum ‘ext‘ default_sToc)
```

We also redefine the synthesised *html* attribute for the header node, using its *headerNum* as a value for `id` in the HTML tag. This gives us a navigation mechanism within the HTML document. We omit the implementation since it closely resembles the *html* rule defined in section 3. We also do not show the construction of a new semantic record *semHtml''* with *mkDoc* since it is similar to *semHtml'* in section 3, except for the addition of the newly defined rules.

We now have all the building blocks to create the new conversion tool:

```
latex2html'' :: String → String
latex2html'' = buildConverter (gLatex semHtml'' +>> gLatexToc semHtmlToc) html
```

The combinator `+>>` composes grammar fragments, such that its second argument extends the grammar in its first. It just composes two arrows, passing the output of the first one (exported non-terminals) as input to the second one.

4 Conclusions, Related and Future Work

We have shown how our libraries can be used to construct a more flexible and extensible *Pandoc* system. We have shown how to extend the underlying parsers for the input language, how to extend the intermediate representation, and how to extend and change computations over this intermediate data structure. The consistency of all definitions is done by the Haskell compiler. We foresee a system in which a document may come with references to the definitions of used document elements (like DTD's) *including* their semantics, i.e. how these elements are to be formatted in combination with other elements.

We heavily lean of the Typed Transformations of Typed Abstract Syntax [1] technique in realising this. One of the questions which arises is whether such flexibility might have been achieved otherwise. On his website[8] the designer of

[8] `http://johnmacfarlane.net/pandoc/scripting.html`

Pandoc, John Macfarlane, shows how some of the things we are doing using our attribute grammar system may be achieved by some form of scripting, which boils down to constructing the abstract syntax tree of the document, and subsequently applying functions to this tree, before writing the tree out in some specific format. We believe that, although this technique may work for simple transformations, this is not the way to go. Such approaches look simple at first, but become cumbersome to use once many (related) transformations are to be applied. They are inefficient since the tree is to be inspected over and over again, and worse, the programmer has to be aware of all the transformations, what they do to the tree, what information to leave in the tree for further transformations, and where to pick it up in further steps. Once one tries to use the Haskell type system to check for the consistency of this process the types of the intermediate trees change, defeating the whole underlying *Pandoc* philosophy. Of course this problem can be circumvented by storing attributes in the tree in the form of dictionaries, but then we move to the untyped world, where the type system does not guarantee that entries referred to are present. If one wants to resort to such untyped techniques one might look at systems which have been designed to support this approach such as Stratego [2] (see however [4] for an extension of transformation systems with attribute grammar facilities). The fact remains however that in all these approaches the life of the programmer becomes much more complicated because he loses strong typing and has to make the evaluation order and the storage and retrieval explicit, whereas this is implicitly done by the lazy evaluation underlying the attribute grammar approach. In a technical report we show the viability of our approach in implementing a compiler for the language Oberon0 in a stepwise fashion [9]. Other systems pursuing solutions along these lines are Kiama [6] which uses Scala as the (host) implementation language and Silver [8].

Once we have introduced a lot of attributes, the tree structures may grow, and accessing the individual attributes may start to add to the overall cost. In [12] we have described how the `AspectAG` code we have shown may be generated by the Utrecht University Attribute Grammar Compiler (*uuagc*) from a less verbose format. Another option which becomes available that way is to group attributes such that they can be accessed faster [12]. The *uuagc* compiler can also read a large collection of attribute grammar fragments, analyse the overall dependencies and generate the tree-walk evaluators which have to be constructed by hand in the more explicit approaches. In this way we can easily generate a very fast compiler for the document type at hand.

Although we have hardly used the full power of the attribute grammar formalism we want to mention that for many kinds of computations over trees they are the tool of choice: *attribute grammars form a domain specific language for describing computations over trees*, where we do not have to limit ourselves to non-circular grammars at all when we use an lazy evaluated underlying language. In the online computation of pretty printed documents we essentially use lazy evaluation to be able to evaluate a circular attribute grammar; something which is not easily transformed into an explicitly scheduled version [7].

References

1. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed abstract syntax. In: TLDI 2009: Proceedings of the 4th International Workshop on Types in Language Design and Implementation, pp. 15–26. ACM, New York (2009)
2. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates. Ph.D. thesis, Utrecht University, Utrecht, The Netherlands (January 2008)
3. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. 37(1-3), 67–111 (2000)
4. Kats, L., Sloane, A., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 142–157. Springer, Heidelberg (2009), `http://dx.doi.org/10.1007/978-3-642-00722-4_11`
5. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proc. of the 2004 Workshop on Haskell, pp. 96–107. ACM Press (2004)
6. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In: Proc. of the Ninth Workshop on Language Descriptions, Tools, and Applications (March 2009)
7. Swierstra, S.D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming 19(01), 1–16 (2009)
8. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Sci. Comput. Program. 75(1-2), 39–54 (2010)
9. Viera, M.: First Class Syntax, Semantics and Their Composition. Ph.D. thesis, Utrecht University, Department of Information and Computing Sciences (2013)
10. Viera, M., Swierstra, S.D., Dijkstra, A.: Grammar Fragments Fly First-Class. In: Proc.of the 12th Workshop on Language Descriptions Tools and Applications, pp. 47–60 (2012)
11. Viera, M., Swierstra, S.D., Lempsink, E.: Haskell, Do You Read Me?: Constructing and composing efficient top-down parsers at runtime. In: Proc. of the First Symposium on Haskell, pp. 63–74. ACM, New York (2008)
12. Viera, M., Swierstra, S.D., Middelkoop, A.: UUAG Meets AspectAG. In: Proc. of the 12th Workshop on Language Descriptions Tools and Applications (2012)
13. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell. In: Proc.of the 14th Int. Conf. on Functional Programming, pp. 245–256. ACM, New York (2009)