Expand: Towards an extensible Pandoc system

An application of extensible compiler construction in Haskell

Jacco Krijnen  Doaitse Swierstra  Marcos Viera

Department of Computer Science, Utrecht University
Utrecht, The Netherlands

Instituto de Computación, Universidad de la República
Montevideo, Uruguay

January 4, 2014
Introduction

We can write documents using different markup languages:

- Latex (.tex)
- markdown (.md)
- Word (.doc)
- mediawiki
- ...

Usage might depend on syntactic preference, language features or purpose.
Introduction

All formats have their own way of describing document elements.

<table>
<thead>
<tr>
<th>document element</th>
<th>HTML</th>
<th>Latex</th>
<th>Markdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>bold text</td>
<td><code>&lt;b&gt;...&lt;/b&gt;</code></td>
<td><code>\textbf{...}</code></td>
<td><strong>...</strong></td>
</tr>
<tr>
<td>header</td>
<td><code>&lt;h2&gt;...&lt;/h2&gt;</code></td>
<td><code>\subsection{...}</code></td>
<td>##...</td>
</tr>
<tr>
<td>paragraph</td>
<td><code>&lt;p&gt;...&lt;/p&gt;</code></td>
<td></td>
<td>Blank lines</td>
</tr>
<tr>
<td>url</td>
<td><code>&lt;a href=&quot;...&quot;&gt;...&lt;/a&gt;</code></td>
<td><code>\url{...}{...}</code></td>
<td><code>[...](...)</code></td>
</tr>
</tbody>
</table>
Introduction

Problem: We have written a document in one language and would like to convert it to another, without doing all the formatting again.
Introduction (Pandoc)

Pandoc is open-source software that allows for automatic conversion between different types of markup formats.

- Supports many types of input and output formats
- Also provides its functionality as a Haskell library
Introduction (Pandoc)

Pandoc is open-source software that allows for automatic conversion between different types of markup formats.

- Supports many types of input and output formats
- Also provides its functionality as a Haskell library
- Architecture:
  
  Readers that can parse specific markup file formats
  
  Abstract Syntax Tree an intermediate data structure to describe a general markup document
  
  Writers that can unparses (or write) to a target markup file format
Introduction (Problem)

Unfortunately, Pandoc’s readers, intermediate type and writers are not extensible.

We propose expand, a Haskell library, in which all of the three components can be described in a modular way, with the possibility to extend and reuse different components.
Contents

Extensibility

Implementing expand: Core Latex to HTML

Abstract Syntax
Grammar - Core Latex
Semantics - HTML
Composing the Tool

Extending components

Numbered Headers
Table of Contents

Conclusion
1. Extensibility
Suppose you want to extend the formatting of markdown.
Suppose you want to extend the formatting of markdown.

One should be able to
Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type

\[\text{§1}\]
Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components
Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components
3. combine the extensions with the existing components
Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components
3. combine the extensions with the existing components

Furthermore, the Haskell type system should verify that the composition is
Extensibility

We base our solution on the following Haskell libraries

- `murder` for defining parser using grammar fragments
- `AspectAG` for describing semantics using attribute grammars rules.

2. Implementing expand: Core Latex to HTML
Example input & output

\section{Introduction}

\begin{paragraph}
Iam id ipsum absurdum, \textbf{maximum} malum neglegi. Ut id aliis narrare gestiant?
\end{paragraph}

\subsection{subintroduction}

\begin{paragraph}
Sic enim censent, oportunitatis esse beate vivere. A primo, ut opinor, animantium ortu petitur origo summi boni.
\end{paragraph}
\section{{Introduction}}
\begin{paragraph}
Iam id ipsum absurdum, \textit{maximum} malum neglegi. Ut id aliis narrare gestiant?
\end{paragraph}

\subsection{{subintroduction}}
\begin{paragraph}
Sic enim censent, oportunitatis esse beate vivere. A primo, ut opinor, animantium ortu petitur origo summi boni.
\end{paragraph}

<h1>Introduction</h1>
<p>Iam id ipsum absurdum, <b>maximum</b> neglegi. Ut id aliis narrare gestiant?</p>

<h2>subintroduction</h2>
<p>Sic enim censent, oportunitatis esse beate vivere. A primo, ut opinor, animantium ortu petitur origo summi boni.</p>
2.1 Abstract Syntax
module Declarations.CoreLatex where

data Document = Document { blocks :: BlockL } deriving Show
module Declarations.CoreLatex where

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)

Figure: A datatype for the AS
module Declarations.CoreLatex where

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_itallInl :: InlineL }
                         deriving (Show)

Figure: A datatype for the AS
module Declarations.CoreLatex where

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"])

Figure: A datatype for the AS
$ (deriveAG "Document")

Generates the necessary labels and types to be used in the attribute grammar fragments.

$ (deriveLang "Doc" ["Document", "BlockL", "Block", "InlineL", "Inline"])  

Generates a record with the appropriate semantic function types, to be used by the parser.
2.2 Grammar - Core Latex
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* "}"

Figure: The EBNF for our input language
module Grammars.CoreLatex where

import Declarations.CoreLatex

gLatex sem = proc () → do
  rec
    doc ← addNT ≺ [ (pDocument sem) blocks ]
    blocks ← addNT ≺ pFoldr (pBlockL_Cons sem, pBlockL_Nil sem)
      [ block ]
    block ← addNT ≺ [ head ] <|> [ par ]
...

Each production is expressed using the idiom brackets\(^1\) \([\ ]\) and \(\|\) (\texttt{iI} and \texttt{Ii} in Haskell code)

\(^1\)http://www.haskell.org/haskellwiki/Idiom_brackets
...  
\[
\begin{align*}
par & \leftarrow \text{addNT} \prec \text{addNT} \prec (p\text{Paragraph sem}) \\
& \\begin{cases} 
\"\begin{\" \{ " \"paragrapgh" \} \"
\text{inls}
\"\end{\" \{ " \"paragrapgh" \} \}
\end{cases} \\
head & \leftarrow \text{addNT} \prec \text{let} \ h(x, \text{name}) = \text{addNT} \prec (p\text{Header sem} \ x) \ \"\begin{\" \text{name} \\
\"\end{\" \text{name} \\
\"\} \}
\text{inls} \\
\} \}
\text{headers} & = [(1, "section") \\
& \quad , (2, "subsection") \\
& \quad , (3, "subsubsection")]
\text{in} \ \text{foldr1 (<|>) (map h headers)}
\end{align*}
...
...  

```latex
\textbf{inls} \leftarrow \texttt{addNT} \prec \texttt{pFoldr (pInlineL\_Cons sem, pInlineL\_Nil sem)} \\
\texttt{inline}
```

```latex
\textbf{inl} \leftarrow \texttt{addNT} \prec \texttt{(pBold sem)} "\\textbf{{" \texttt{inlineL }"}}" \\
\texttt{inlineL} < | > \texttt{(pItalics sem)} "\\textit{{" inlineL "}}" \\
\texttt{inlineL} < | > \texttt{(pPlain sem)} "\\plain{{" (someExcept "\\&%\$\#\_\{\}\~\") ""}) \\
\texttt{exportNTs} \prec \texttt{exportList document ( export cs\_document doc} \\
\texttt{export cs\_blockL blocks} \\
\texttt{export cs\_paragraph par} \\
\texttt{export cs\_header head} \\
\texttt{export cs\_inline inl} \\
\texttt{export cs\_inlineL inls)
```

Figure: Our EBNF encoded as a series of grammar transformations
2.3 Semantics - HTML
We want to compute a *String* from the abstract syntax tree: the HTML text. Therefore we introduce a *synthesized attribute* `html`.

```haskell
module Semantics.HTML where

import Declarations.CoreLatex

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

Using *AspectAG*, we can now define a rule per data constructor, describing how to compute this *String*. 
document_html = ...
blockLnil_html = ...
blockLcons_html = ...

header_html = syn html$
    do level ← at ch_level_header
        inls ← at ch_inlines_header
        return $"<h" ++ show level ++ ">"
            ++ inls # html
            ++ "</h" ++ show level ++ ">"
            ++ 

bold_html = syn html$
    do inls ← at ch_inlines_boldInl
        return $"<b>"
            ++ inls # html
            ++ "</b>"

...
We can now generate the record of semantic functions

\[
\text{semHtml} = \text{mkDoc blockLcons\_html}
\]
\[
\quad \text{blockLnil\_html}
\]
\[
\quad \ldots
\]
\[
\quad \text{paragraph\_html}
\]
\[
\quad \text{plain\_html}
\]

... to be used by the parsers:
We use a deforestated approach, i.e. the Tree never comes into existence.
2.4 Composing the Tool
module Converter where

import Grammars.CoreLatex (gLatex)
import Semantics.HTML (semHTML, html)
import Expand_Utils (buildConverter)

latex2html :: String → String
latex2html = buildConverter (gLatex semHtml) html
3. Extending components
3.1 Numbered Headers
We want to extend the HTML generation in a way that headers are numbered.
We want to extend the HTML generation in a way that headers are numbered.
Numbered Headers

We introduce a chained attribute \( cHeaderNum \), which is threaded through the tree in a \( \text{State Monad} \) like fashion. We keep a local copy of the value in every header by introducing the \( headerNum \) attribute (so that it is available for future extension).

\[
\text{module } \text{Semantics\.General\.NumberedHeaders } \text{where}
\]

\[
\text{import } \text{Declarations\.CoreLatex}
\]

\[
\text{\$ (attLabels ["cHeaderNum", "headerNum"])}
\]
Numbered Headers

§3.1

We introduce a chained attribute \texttt{cHeaderNum}, which is threaded through the tree in a \textit{State Monad} like fashion. We keep a local copy of the value in every header by introducing the \texttt{headerNum} attribute (so that it is available for future extension).

\begin{verbatim}
module Semantics.General.NumberedHeaders where

import Declarations.CoreLatex

$(attLabels ["cHeaderNum", "headerNum"])

We will model the header number as a \texttt{[Int]}, and introduce two utility functions.

\begin{verbatim}
updateHeaderNum :: Int \rightarrow [Int] \rightarrow [Int]
updateHeaderNum level par = zipWith (+) par' (zeros ++ [1])

where par' = par ++ repeat 0
zeros = replicate (level - 1) 0

formatHeaderNum :: [Int] \rightarrow String
formatHeaderNum = intercalate "." \circ map show
\end{verbatim}
\end{verbatim}
cHeaderNum\_NTs = nt\_BlockL .\* . nt\_Block .\* . hNil

default\_cHeaderNum = chain cHeaderNum cHeaderNum\_NTs
document\_cHeaderNum = inh cHeaderNum cHeaderNum\_NTs $
do return (ch\_blocks .\= . ([] :: [Int]) .\* . emptyRecord)

header\_headerNum = loc headerNum$
do lhs ← at lhs
 level ← at ch\_level\_header
 return $ updateHeaderNum level (lhs ≠ cHeaderNum)

header\_cHeaderNum = syn cHeaderNum$
do loc ← at loc
 return $ loc ≠ headerNum

Note that the attribute rules are independent of the target language.
Numbered Headers

§3.1

We still have to change the HTML output. We use the *synmodM* function to define a modification of a rule.

```haskell
module Semantics.HTML.NumberedHeaders where
import Declarations.HTML
import Semantics.HTML

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"
```
Numbered Headers

§3.1

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

\[
\text{semHtml}' = \text{mkDoc}
\]

\[
(\text{default}_\text{cHeaderNum} \ 'ext' \ \text{blockLcons}_\text{html})
\]

\[
(\text{default}_\text{cHeaderNum} \ 'ext' \ \text{blockLnil}_\text{html})
\]

\[
\text{bold}_\text{html}
\]

\[
(\text{document}_\text{cHeaderNum} \ 'ext' \ \text{document}_\text{html})
\]

\[
(\text{header}_\text{headerNum} \ 'ext' \ \text{header}_\text{cHeaderNum}
\]

\[
'ext' \ \text{header}_\text{html}'
\]

\[
'ext' \ \text{header}_\text{html})
\]

\[
\text{inlineLcons}_\text{html}
\]

\[
\text{inlineLnil}_\text{html}
\]

\[
\text{italics}_\text{html}
\]

\[
(\text{default}_\text{cHeaderNum} \ 'ext' \ \text{paragraph}_\text{html})
\]

\[
\text{plain}_\text{html}
\]
Our alternative converter now becomes:

\[
\text{latex2html'} :: \text{String} \rightarrow \text{String} \\
\text{latex2html'} = \text{buildConverter \ (gLatex \ semHtml')} \text{ html}
\]
3.2 Table of Contents
A table of contents requires extensions in all three components.

- **Grammars:**
  - Core Latex
  - Core Latex + TOC

- **Declarations:**
  - Document
  - Document + TOC

- **Semantics:**
  - HTML
  - HTML + TOC
The abstract syntax needs support for a TOC node.

```
module Declarations.CoreLatex.Toc
import Declarations.CoreLatex

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

The function `deriveLang` will also produce a new record type containing the semantic function of the `Toc` production.
Our latex language needs an additional construct:

```
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" * "}"```

Our latex language needs an additional construct:

\[
\begin{align*}
document & ::= \text{block} \ast \\
\text{block} & ::= \text{paragraph} \\
& \quad | \text{header} \\
& \quad | \"\text{tableofcontents}\" \\
\text{paragraph} & ::= \"\texttt{\begin{paragraph}} \text{inline} \ast \texttt{\end{paragraph}}\" \\
\text{header} & ::= \texttt{\section} \text{inline} \ast \\
& \quad | \texttt{\subsection} \text{inline} \ast \\
& \quad | \texttt{\subsubsection} \text{inline} \ast \\
\text{inline} & ::= \texttt{\plain} \text{text} \\
& \quad | \texttt{\textbf} \text{inline} \ast \\
& \quad | \texttt{\textit} \text{inline} \ast
\end{align*}
\]
Since the abstract syntax tree now supports a table of contents, we can extend the grammar for the latex-like language:

```plaintext
module Grammars.CoreLatex.Toc
import Grammars.CoreLatex
import Declarations.CoreLatex
import Declarations.CoreLatex.Toc

gLatexToc sem = proc imported →
  do
    let block = getNT cs_block imported
    addProds ≺ (block, \( pToc sem \) "\\tableofcontents")
    exportNTs ≺ imported
```
We model the table of contents as a `([[Int], String)])`.

```haskell
module Semantics.General.Toc where
import Declarations.CoreLatex

attsLabels ["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)]
```
Using the inherited attribute `toc` we can now format it using an appropriate `html` rule.

```haskell
module Semantics.HTML.Toc where

import Semantics.General.Toc

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
```

The `html` formatting of a header is changed so that it contains an id attribute (omitted)
We now have all the building blocks to create the new conversion tool:

\[
\text{latex2html}'' :: \text{String} \rightarrow \text{String} \\
\text{latex2html}'' = \text{buildConverter} \ ( \ g\text{Latex} \ sem\text{Html}'' \\
+>> \ g\text{LatexToc} \ sem\text{HtmlToc}) \ html
\]
\tableofcontents
\section{Introduction}
\begin{paragraph}
Iam id ipsum absurdum, \textbf{maximum} malum neglegi. Ut id aliis narrare gestiant? 
\end{paragraph}

\subsection{subintroduction}
\begin{paragraph} ...
\end{paragraph}

\subsection{subintroduction}
\begin{paragraph} ...
\end{paragraph}
\tableofcontents
\section{Introduction}
\begin{paragraph}
Iam id ipsum absurdum, \textbf{maximum} malum neglegi. Ut id aliis narrare gestiant? 
\end{paragraph}

\subsection{subintroduction}
\begin{paragraph} ...
\end{paragraph}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
1 | Introduction & 1.1 | subintroduction \\
\hline
2 | Conclusion \\
\hline
\end{tabular}
\end{table}

<h1 id="[1]">1 Introduction</h1>
<p>Iam id ipsum absurdum, <b>maximum</b> neglegi. Ut id aliis narrare gestiant? </p>

<h2 id="[1,1]">1.1 subintroduction</h2>
<p>...</p>
4. Conclusion
Conclusion

- We have shown how to build an extensible document formatting system using the {	exttt{murder}} and {	exttt{AspectAG}} libraries.

All code can be found at [http://hackage.haskell.org/package/expand](http://hackage.haskell.org/package/expand).
Conclusion

- We have shown how to build an extensible document formatting system using the murder and AspectAG libraries.
- The Haskell type system validates the composition of all the extensions
  - Grammar fragments state in their type which non terminals they require and expose
  - All usages of non-terminals are guaranteed to point to valid productions
  - AG rules state in their type which attributes should be present and which attribute they define

All code can be found at http://hackage.haskell.org/package/expand.
Conclusion

- We have shown how to build an extensible document formatting system using the murder and AspectAG libraries.
- The Haskell type system validates the composition of all the extensions
  - Grammar fragments state in their type which non terminals they require and expose
  - All usages of non-terminals are guaranteed to point to valid productions
  - AG rules state in their type which attributes should be present and which attribute they define
- By the modular nature of the components we can reuse aspects, like the Table of Contents.

All code can be found at http://hackage.haskell.org/package/expand.
header_sToc

:: (HasField (Proxy (Ch_inlines_header, [Inline]))) chi r,
HasField (Proxy Att_headerNum) l t,
HasField (Proxy Att_sInlStr) r t1,
HExtend (Att (Proxy Att_sToc) [(t, t1)]) sp sp' \Rightarrow
Rule l ho chi par l1 ho1 ic sp l1 ho1 ic sp'

header_sToc = syn sToc $

\textbf{do} \quad \text{loc} \leftarrow \text{at} \ \text{loc}
\quad \text{inls} \leftarrow \text{at} \ \text{ch_inlines_header}
\quad \text{return} \ [(\text{loc} \# \text{headerNum}, \text{inls} \# \text{sInlStr})]