BlaTeX

Rushi Shah
TJHSST

Abstract

Write your blog in LaTeX.

Introduction

Markdown and HTML are the standard tools used to write your every day tech blog with. But they have pretty weak support for embedding mathematical formulas, and are not conducive to writing for an extended period of time. Plus, they aren't even Turing complete! So instead of writing posts in Markdown, my project allows writers to create posts in LaTeX (a powerful markup language) and generate a static-site that can be deployed to any hosting service (like TJ's servers).

Background

Typically, static-sites like blogs are built with a tool called Jekyll. This Ruby project compiles Markdown posts into a static HTML site. BlaTeX is similar, it is a static-site compiler written in Haskell that compiles LaTeX posts into a deploy-able site. The following description has been edited to highlight the minor differences between Jekyll and BlaTeX :

~~Jekyll~~ BlaTeX is a simple, blog-aware, static site generator. It takes a template directory containing ~~raw text files in various formats~~ LaTeX files~~, runs it through a converter (like Markdown) and our Liquid renderer,~~ and spits out a complete, ready-to-publish static website suitable for serving with your favorite web server. ~~Jekyll also happens to be the engine behind GitHub Pages, which means~~ you can use ~~Jekyll~~ BlaTeX to host your project's page, blog, or website from GitHub's servers for free.

Another popular content-management system for blogging is Wordpress. However, Wordpress posts are edited in the Wordpress Administration Panel as "What you see is what you get". Neither Markdown nor LaTeX are "What you see is what you get", and thus Wordpress caters to a completely different audience (a group of mostly non-technical authors).

Development & Techniques

Requirements. This command line utility requires the Haskell Platform and LaTeX. The Haskell Platform includes GHC (a Haskell compiler) and Cabal, which is Haskell's primary package manager. LaTeX is needed to build LaTeX posts into their resulting PDFs. After the Haskell Platform has been installed, installing BlaTeX is simple:

```
> cabal install blatex
```

Overview. The project is a command line utility that is written in Haskell. Haskell is a statically-typed functional programming language that makes heavy use of the REPL. Libraries were tested/experimented with independently in their own files. After the necessary functions were written, they were integrated into the final product. The finished project was assembled into an executable file and deployed to Haskell's package manager (Hackage).

Limitations. One potential limitation of this project is that it only compiles links to PDF files. Because LaTeX can't generate HTML, the blog will only be a series of interlinked PDF files rather than a series of interlinked HTML files. This is not necessarily a limitation, but it may interrupt the status quo of typical blogs made of HTML pages.

In order to parse the user's LaTeX files into an abstract syntax tree that can be consumed by Haskell, this project uses a Haskell package called `HaTeX`. This is a wonderful package (most of the time). The problem arises in a specific parsing instance. In LaTeX, a math environment is defined between two dollar-signs (`$MATH GOES HERE$`). In order to use dollar-signs as actual dollar-signs (like "You owe me $5"), you use an escape character before the symbol (\). This is an exception to the math environment rule. However, HaTeX is not programmed to catch this exception gracefully. It matches pairs of dollar-signs and treats everything in between as math. If there are an odd number of dollar signs, or only one (even if they are escaped properly) HaTeX will just flat out fail to parse the entire file and thus BlaTeX will similarly ignore the post.

Dates are formatted in `DD MONTH YEAR` which contributes to the minimalist, classy theme of PDF blog posts; this date format is also used in the internal implementation. However, this becomes an issue if multiple posts are authored on same day because they will not be displayed in the correct order.

Research Theory and Design Criteria

Error Handling. Error handling was one of the most complex aspects of this project. Haskell is a strongly typed language and utilizes Type Classes for a large portion of functionality to produce elegant and expressive code. Series of functions that could potentially create errors were chained together with an instance of an Applicative Functor. Originally the Maybe Applicative was used, but in order to

produce meaningful error messages for the user, I switched to the Either Applicative. A simplified example of code used to handle some potential errors is as follows:

```
createPost s t = Post <$> pure s <*> title <*> author <*> date <*> pure t
  where
    date = (getCommandValue "date" t)
    author = (getCommandValue "author" t)
    title = (getCommandValue "title" t)
```

In this code, the `getCommand` function will either find the command within the abstract syntax tree, or the given command was never used. If the command was never used, no post can be created because the requisite information was missing. This series of function calls are tied together with `<*>`, the "splat" operator on Applicative instances. If one computation fails, the ultimate computation will also fail.

Input/Output. Because Haskell is a purely functional programming language, no side effects are permitted. This makes input/output rather difficult, and thus `Monad`'s are used to simulate the desired side effects. However, ubiquitous use of Monads defeats the purpose of the purely functional nature of Haskell. Thus, the input-output code was confined to a single module where almost all effect-ful computations were handled. The main function was included here that glued together all pure functions. This function included code to automatically initate the proper file structure and skeleton files for a new user through system commands. It also included code for the building process. The output of this process is:

```
Getting directory contents
Turning directory contents into posts
All posts are well formed
Turning posts into an HTML element
Reading the layout file
Inserting HTML element into layout file
Writing resulting file into index.html
Success building!
```

If an error is raised at any point in the process, the process will halt and display the error message for the user to troubleshoot.

Developmental Procedures

In general terms: a directory containing posts is examined by BlaTeX, each post is parsed into an abstract syntax tree, information from this tree is parsed into a Haskell Post object, each post is stored into a list, the list is sorted by the date, and each object is converted into it's corresponding HTML, and this HTML is outputted into an output file at the correct location.

The `Post` Algebraic Data Strucure. Haskell doesn't have "objects" per-se, but rather it has "algebraic data types". They are kind of similar, so for the purposes of this paper you can consider them equivalent. The final definition for the Post was:

```
data Post = Post {
  fileName :: String
  , postTitle :: String
  , postAuthor :: String
  , postDate :: DateTime
  , syntaxTree :: LaTeX
    }
    deriving (Eq, Show)
```

This basically defines the structure of a Post based on each attribute it would have and what type that attribute would have.

BlazeHTML. The first part of the project that I tacked was generating HTML elements that would need to be inserted into the final file. Each post object would generate the following LI element:

```
<li class='blog-post'>
    <a class='post-link' href='posts/example-post.pdf'>
        Example Post
    </a>
    <div class='post-date'>
        1 January 2015
    </div>
</li>
```

HaTeX. The second part of the project required that I read in LaTeX files for the posts and use them to extract certain meta-information about each post. This included the title of the post (inserted into the `.post-link` tag) and the date of the post (used to sort posts and displayed in the `.post-date` tag).

TagSoup. The user supplies a layout file which defines the format and style of their blog. BlaTeX finds the line `<ul id='blog-posts'></ul>` and inserts all the posts at that point. In order to do so, the project reads in an HTML file and derives it's abstract syntax tree with a package called TagSoup. TagSoup converts an HTML file into a list of HTML elements, and the HTML generated with BlazeHTML is inserted at the specified point.

Dates. Support for dates was one of the final undertakings addressed. The Data.Dates package was used to parse the LaTeX dates. This allowed me to sort the list of posts chronologically.

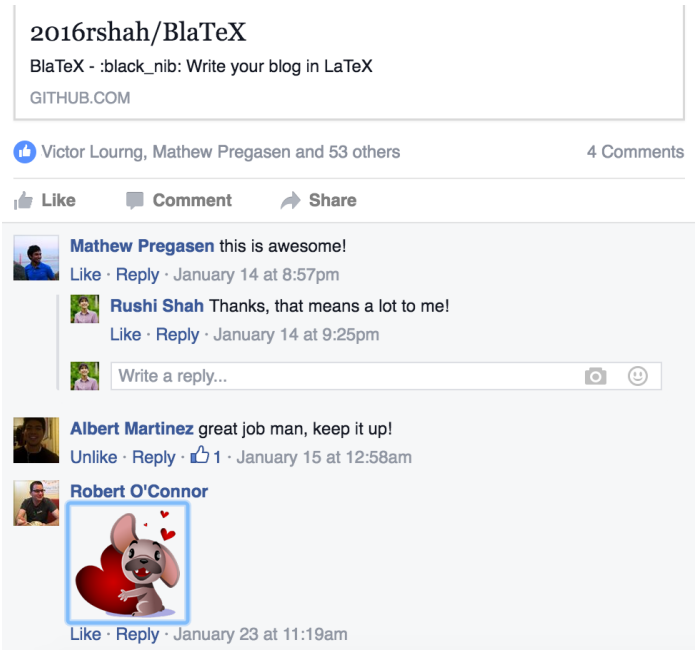
Internal Testing and Analysis

This tool was written in Haskell, a statically-typed, functional, compiled programming language. The compiled nature filtered out most insidious bugs in the code. Haskell development is also strongly integrated with using a Haskell REPL. Thus, each function was experimented with in the REPL after being written and before being pushed to production.

User Interface Testing and Analysis

In order to test this project, I used it to create and maintain `http://rshah.org/blog`. Through my own use I discovered multiple improvements that could be made to the user interface and experience, which I promptly implemented.

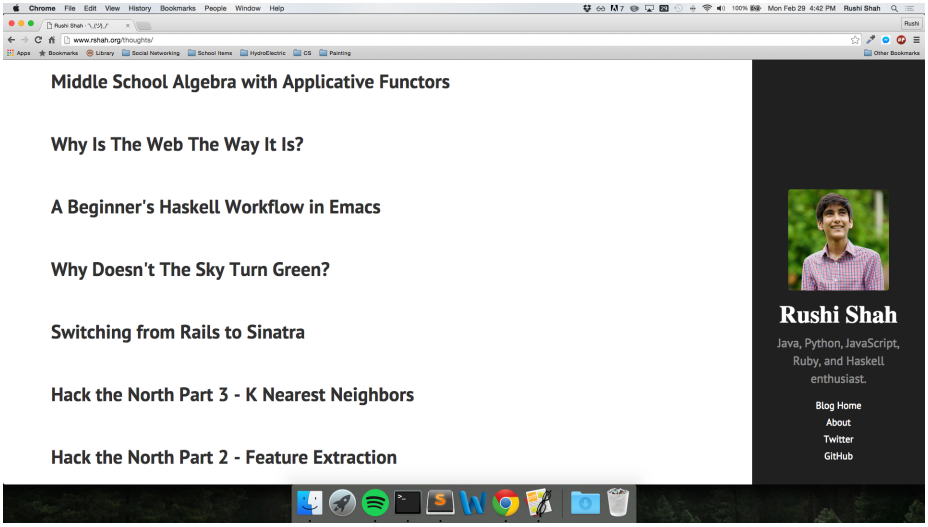
I also posted my project in a Facebook group for functional programming and recieved an overwhelmingly positive response:



It is also an open source project on Github, and met a similarly positive reaction from that community (with 17 Github "stars").

Visual representation of Results

My blog at `http://rshah.org/blog` is a visual representation of a static site compiled with BlaTeX.



Results, Discussion, and Conclusion