Arun Kulshreshtha
CS 240H Final Report
March 18th, 2016

HaskML: HTML Combinators for Haskell

Introduction

My project, HaskML (a portmanteau of "HTML" and "Haskell"), is a simple HTML combinator library for Haskell. The goal of this library is to provide a simple but robust API for creating dynamic HTML content in Haskell, that emphasizes building documents through the composition of small components rather than string concatenation or templating. The source code is available at https://github.com/kulshrax/haskml.

Motivation

This project was initially inspired by XHP, an extension to PHP developed as part of Facebook's Hack programming language (https://docs.hhvm.com/hack/XHP/introduction). XHP augments PHP's syntax to allow programmers to write fragments of markup as literals in the language. An example XHP syntax can be seen here:

```
$my_xhp_object = <p>Hello, world</p>;
```

These literals are parsed into XHP components, which can then be composed into larger structures, and ultimately web pages. The approach to building web pages afforded by XHP is certainly more elegant than the previous alternative used at Facebook, which involved simply representing HTML fragments as strings and concatenating them together throughout the code. It is arguably even more elegant than traditional templating systems, which still effectively treat markup as strings to be substituted into a template, which is itself a larger string.

However, to get this sort of nice syntax and functionality into PHP, the authors of XHP needed to write a language extension. In contrast, in a language like Haskell, which offers a much greater degree of expressivity and concise syntax, one could imagine creating something akin to a DSL for HTML markup as a library, rather than as a language extension. This is what HaskML attempts to do.

One thing to note is that this idea has been implemented before. There is already an HTML combinator library for Haskell called BlazeHTML (https://jaspervdj.be/blaze/). However, I chose to try to implement this anyway, as it seemed like something that would both be a good learning experience -- in particular, creating a clean, compositional API would require mastering the facets of Haskell that allow such clean abstractions to be possible. While my library is ultimately not as fast or feature-rich as BlazeHTML, it provides a simple yet practical example of how one can build a combinator library in Haskell.

A Tour of HaskML

The main API exposed by HaskML is a series of combinators defined in the files Tags.hs and Attributes.hs. These combinators correspond to the tags and attributes listed in WHATWG's HTML5 specification (https://html.spec.whatwg.org/multipage/). If one enters one of these combinators into GHCi, one will see the following behavior:

Input: `html :: HtmlM a -> HtmlM b`
Output: `<html></html>`

Note that the combinator html shown above is a function. However, these combinators have custom Show instances defined for them that render the corresponding markup when the combinators are printed. As expected, combinators can be combined using function composition:

Input: `html . body`
Output: `<html><body></body></html>`

Furthermore, one can use the `(!)` operator, combined with the attribute combinators defined in Attributes.hs, to add attributes to the resulting HTML elements. (This syntax was inspired by BlazeHTML.) In practice, since many of these combinators have names that conflict with functions in the Prelude, they should in practice be imported qualified. For the remainder of this paper, assume the following preamble to any code examples:

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Tags as H
import qualified Attributes as A
```

Note that we import Tags as `H`, rather than `T`. This is because in practice, one would also import the Data.Text module alongside HaskML, and this module is commonly imported as `T`. HaskML uses `Data.Text` objects rather than normal Haskell `String` type for all of its data since the former, being a packed array of UTF-16 code points, is a much more efficient representation of (potentially Unicode) textual data than a list of characters. With this in mind, we return to our example:

Input: `H.html . H.body ! A.bgcolor “#fff” $ “Hello, world!”`
Output: `<html><body bgcolor=”#fff”>Hello, world!</body></html>`

In this example, it looks like “Hello, world!” is simply a Text object. However, it turns out that in this case, its type is actually `HtmlM a`. This is because when a string literal is encountered in a combinator expression, it is actually parsed by the library into an HtmlM object, which is then treated as a combinator within the expression.

This is made possible by the OverloadedStrings language pragma, which HaskML heavily relies upon. Why is this useful? It turns out this makes it possible to write fragments of HTML as string literals, and the library will correctly parse them into the data structure that one would get if one had constructed it using HaskML’s combinators. As such, one can mix the combinator syntax with regular HTML syntax (within a string literal).

Input: `H.div $ “<a>A link.</a>” ! A.href “haskell.org”`
Output: `<div><a href=”haskell.org”>A link.</a></div>`

One nice thing about this is that it is about as close as one can get to the “bare” markup that seen in XHP without actually extending the actual parser of the language to include HTML parsing. Compared to most other languages, this is a place where Haskell really shines, enabling very concise syntax (literally just quotation marks for the string literal) to be implemented by a library.

Of course, in practice HTML documents involve a lot of nesting. HaskML takes advantage of Haskell’s do notation to provide a nice syntax for this:

Input:

```
H.div $ do
      “<ul></ul>” $ do
            H.li “Item 1”
            H.li “Item 2”
```

Output: `<div><ul><li>Item 1</li><li>Item 2</li></ul></a>`

How is this notation possible? It turns out that as the name implies, HtmlM is an instance of Monad. However, this is a case where taking advantage of Haskell syntactic sugar for monads requires playing some games with the type system. In particular, in order for a type to be an instance of monad, its constructor must have kind * -> *, which is to say, it must take a type parameter.

HaskML’s internal representation of an HTML AST -- namely, the Html type defined in Html.hs -- does not take any type parameters, since it doesn’t require any sort of polymorphism. To allow users to use the do notation to combine HTML fragments, HaskML defines an HtmlM a type, whose type parameter literally does nothing:

```
newtype HtmlM a = HtmlM { getHtml :: Html }
```

This allows us to define an instance of Monad for our combinators. However, as one might expect, this “monad” is more of a hack than anything. In particular, since it doesn’t actually contain anything, the monad doesn’t obey the usual monad laws, which is admittedly rather ugly. However, it turns out that in practice, many Haskell DSLs are implemented this way (notably, BlazeHTML makes a similar “hack” to utilize the do notation), and this is largely a consequence of the fact that the nice do-notation syntactic sugar is only available for monads. If there were equally nice syntactic sugar for, say, monoids, then that abstraction would probably be more appropriate in this case. (Note that Html and HtmlM objects are in fact monoids, but composing them using the `(<>)` operator makes the structure of our templates further removed from the structure of the resulting HTML -- something that HaskML tries to avoid.)

Using HaskML’s combinators and syntax, one can construct “templates” for web pages by simply creating functions that return HtmlM combinators, with input variables (which could be Text, or more combinators) getting plugged into the correct places. Here is an example of what a complete web page template might look like:

```
template name content url items =
    let title = name <> "'s Page" in
        H.html $ do
            H.head $ do
                H.meta ! A.charset "UTF-8"
                H.title title
            H.body $ do
                "<h1></h1>" $ title
                H.div $ do
                    "<a>A link.</a>" ! A.href url
                    H.ul $ mapM H.li items
                    "<p>Content appears below:\n</p>" $ do
                        text content
```

As you can see, the syntax and structure of this template looks fairly close to that of the underlying HTML, with the mixing of raw HTML fragments and Haskell combinators, all together in a nested structure provided by Haskell’s do notation. Overall, this serves not only as a useful tool for web programmers, but also as an illustrative example of how Haskell can allow programmers to build clean DSLs for a variety of problem domains.

Future Work

Due to time constraints this quarter, I wasn’t able to incorporate all of the functionality that I wanted to into HaskML. In particular, although the library currently makes it pretty easy to build HTML templates using the provided combinators, it doesn’t have a good way of querying those combinators to extract and/or modify Nodes or subtrees that match certain criteria. A proposed method of doing this would involve adding CSS-style selectors

That said, I de-prioritized this functionality during development because of the way the library is intended to be used. Since the most common use case for HTML combinators would be to start with simple elements and combine them into larger, more complicated structures, it seems like a program built using HaskML would have a bottom-up structure, in the sense that one would first build small pieces of the HTML and then combine them into a full page. As such, the ability to query for particular bits of HTML is less useful, since theoretically one would already have access to that piece of HTML (as a combinator) at the time one creates it.