The Abstraction Progression of BlaTeX

Rushi Shah

31 May 2016

Use the Types, Luke

Chris Done (kind of)

My first significant Haskell undertaking was BlaTeX, which is a static site compiler for LaTeX documents (it is currently used to maintain this blog!). It was my senior research project in high school and my Haskell-fu increased dramatically throughout the year. This became more and more evident throughout the commits on the repository as my code became cleaner, more clear, more functional, more concise, and just flat out better. I learned a lot on the journey of writing BlaTeX, and I used some of the alluring features of Haskell for the first time outside of the curriculum I used to learn them. In this post, I will explore how some of my earlier code for BlaTeX was abstracted away beautifully.

1 Maybe to Either

Initially, as many Haskell beginners do, I used Maybe to represent any failed compuations. This worked beautifully until I tried to build my directory of posts and it didn't work and I had no idea why. It turns out that I was trying to parse a directory, not a set of .tex/.pdf files, but the program didn't tell me that, it just didn't do anything. That's the problem with Maybe: it either works or it doesn't, but if it doesn't work it doesn't give any further information about what went wrong.

Also, one of the libraries I was using did not use Maybe, it used Either. So I wrote the following code to change their Maybe value to an Either value:

```
eToM :: Either a a -> Maybe a
eToM e = case e of
   Left _ -> Nothing
   Right d -> (Just d)
```

Eventually I made the switch to Either in order to give more meaningful errors and so I didn't have to use that atrocity of a function (I cringe just looking at it now, it is destroying so much useful information). It wasn't difficult to change between the two error types, and I am very glad I did.

2 Either Applicative

2.1 Background

Okay so now I had a ton of Either values that I needed to amalgamate into the post algebraic data type:

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

To get the postTitle, postAuthor, etc. I was parsing the LaTeX file, but sometimes the user forgets to include certain commands so I needed to alert them of that. If everything worked I wanted a `Right Post`.

2.2 Implementation

For anyone who has made it through Week 10 of CIS194 knows, this problem is practically screaming for using the `<*>` operator (pronounced the "splat operator"). Given the `getCommandValue` function that returns Either a String representing the failure or a String representing the result, I ended up with code that looks roughly like this:

```
createPost s t = Post `fmap` pure s <*> title <*> author <*> date <*> pure t
  where
    author = getCommandValue "author" t
    title = getCommandValue "title" t
    [...]
```

3 Either Monad

3.1 ParseError versus String

Later on in the project, I decided to implement Dates for the posts to order them chronologically. I found a great library to parse the dates, but obviously not all strings can be parsed as dates, so the library returned an Either as well. I didn't think that would be a problem, I just used the functor instance for either and wrote:

```
(fmap (parseDate time) (getCommandValue "date" t))
:: Either String (Either ParseError String)
```

But now I had a doubly-nested Either with two different types of errors and I didn't know what to do. I gave up and wrote the following function that would handle the nesting of the two Either types:

```
hackyTypeMagic :: Either String (Either b c) -> Either String c
hackyTypeMagic (Left e) = Left e
hackyTypeMagic (Right (Left _)) = Left "Could not parse date"
hackyTypeMagic (Right (Right r)) = Right r
```

To be clear, I actually wrote that `hackyTypeMagic` function verbatim and used it for quite some time: it is still in my commit history.

3.2 The Monadic `join` function

After thinking about it for a while, I was certain that this doubly nested either was a common situation and there must be an abstraction for it. It didn't take me long to find the `join` function in Control.Monad which has the following type signature:

```
join :: Monad m => m (m a) -> m a
```

That type signature looks awfully similar to `hackyTypeMagic` so I tried to plug it in. However, I ran into an issue because ParseErrors are not implicitly Strings, they need to be explicitly converted. Before I would just `show` the ParseError to convert it to a String after everything was said and done, but join only works on two Monads of the same type. So to rectify the error message, I converted a string before joining the two Monads together by writing a new function called `parseAbsoluteDate` (the new function also rectified another smaller issue).

To be clear, all the join function does is fuse two Monads together. The Either instance of Monad means that it will just peel off a layer of Either and return the inside if they are both Right.

3.3 »=

Okay great, so now that everything worked, I ended up with something similar to the following code:

```
where parsedDate = join (fmap (parseDate time) ((getCommandValue "date" t)))
```

But take a look at the type signature for join and fmap:

```
join :: Monad m => m (m a) -> m a
fmap :: (a -> b) -> f a -> f b
```

And therefore:

```
(join fmap) :: (b -> b) -> b -> b
```

Which is ridiculous! I was just applying a function at the end of the day, albeit within a computational context. I was basically trying to peek into the wrapping paper with fmap and then eventually taking off the wrapping paper afterwards with join. Instead, I might as well take off the wrapping paper first and then apply the function directly. I had discovered a usecase for the Monadic bind in the wild!

To refresh your memory, the bind operator (»=) has the following type signature:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

So `m a` corresponds to the `getCommandValue "date" t` (the first either monad produced). Then I have a function (`parseAbsoluteDate :: String -> Either String DateTime`) that takes an unwrapped value and wraps it in a new either monad. And finally that will return the Either value that I was originally looking for to insert into my applicative functor. I.E the code simplifies to:

```
date = (getCommandValue "date" t) >>= parseAbsoluteDate
--compared to
date = hackyTypeMagic (fmap (parseDate time) (getCommandValue "date" t))
```

4 Conclusion

Look, Haskell took me forever to learn. I have been using it for upwards of a year and I still feel like a journey-man at best. And the thing is, even when you think you've got the hang of it and you're using it for your projects, you might not be utilizing the strengths of Haskell in the first place. This isn't to say you shouldn't use Haskell; quite the contrary! All I'm saying is you just have to practice using Haskell and always consider how to make code better if you want to write idiomatic and expressive code. Trust me, its worth it.