TYPE-CHANGING REFACTORINGS IN HASKELL

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Stephen Adams

June 2016

Abstract

This mini-thesis tells you all you need to know about...

Acknowledgements

I would like to thank...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter will introduce the most basic concepts of this thesis. In particular it will discuss, refactoring in general, functional refactoring, and the Haskell programming language. It will also state the contributions of this research and outline the rest of the thesis.

Chapter 2

Background: Refactoring Haskell in HaRe

Chapter 2 is where the development and implementation of HaRe will be discussed. The chapter with cover some of the history or HaRe and the briefly the technology that it was originally developed with. Next it will cover the design and implementation of HaRe currently and its dependencies (in particular ghc-exactprint).

A brief outline of this chapter:

1. The original implementation of HaRe
2. The GHC API
3. Generic Programming (Scrap Your Boilerplate and Strafunski-StrategyLib)
4. GHC-ExactPrint
5. The Current version of HaRe

Chapter 3

Data refactoring in a functional context

This chapter will aim to introduce the concept of a type changing or data refactoring. The concept of a data refactoring is taken from Fowler (1999), however many of these refactorings are not applicable outside of an object oriented context. This research has adapted the idea of a refactoring that changes the datatypes a program uses to fit into the functional paradigm.

This chapter will provide several examples of simple data refactorings for the functional language Haskell. These refactorings include transforming standard lists into Hughes lists (Hughes 1986), introducing a new type synonym, and generalising the Maybe type to the typeclass MonadPlus.

A brief outline of this chapter:

1. Data refactorings in an Object Oriented Context
2. Functional Data refactorings
3. Experience report on developing refactorings with GHC-Exactprint and the GHC API
4. Introducing a Type Synonym

5. Generalising Maybe to MonadPlus

Chapter 4

Generalising Monads to Applicative

The previous chapter introduced the concept of a functional data refactoring and gave two examples, introducing a type synonym and generalising Maybe to MonadPlus. This chapter will cover another generalising refactoring in more depth, rewriting monadic functions to use applicative functors.

In their 2008 functional pearl “Applicative programming with effects" Conor McBride and Ross Paterson introduced a new typeclass that they called Idioms but are also known as Applicative Functors (McBride and Paterson 2008). Idioms provide a way to run effectful computations and collect them in some way. They are more expressive than functors but more general than Monads, further work was done in (Lindley, Wadler and Yallop 2011) to prove that Idioms are also less powerful than Arrows.

Applicative functors were implemented in GHC as the typeclass `Applicative`. An interesting part of the history of GHC is that despite McBride and Paterson proving in their original functional pearl that all monads are also applicative functors, however, GHC did not actually require instances of monad to also be instances of Applicative until GHC’s 7.10.1 release (GHC 2015). Now that every monad must also be an applicative functor there now exists a large amount of code which could be rewritten using the applicative operators rather than the monadic ones.

This chapter will discuss the design and implementation of a refactoring which will

automatically refactor code written in a monadic style to use the applicative operators instead. Section 4.1 is a brief overview of the `Applicative` typeclass's operators, section 4.2 will discuss the applicative programming style and, in general, how programs are constructed using the applicative operators, next, section 4.3 will cover some common applications of this refactoring, section 4.4 will specify the refactoring itself, section 4.5 covers the preconditions of the refactoring, finally section 4.6 outlines other refactorings that may be used in conjunction with the generalising monads to applicative refactoring and some possible variations of this refactoring.

4.1 The Applicative Typeclass

The `Functor` typeclass defines a single function that must be implemented, `fmap`.

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
```

The `fmap` function allows for a function to be applied to the contents of the Functor f. One could think of the functor as a context and `fmap` as a function that allows other functions to run within that context. However, what if you wanted to chain together sequences of commands within that context? This is not possible with just functors since `fmap` does not have the function inside of the functor's context. Sequencing commands will require a more powerful abstraction, applicative functors (O'Sullivan, Goerzen and Stewart 2008).

In Haskell applicative functors are implemented in the `Applicative` typeclass. `Applicative` typeclass declares two functions, `pure` and `(<*>)`. The types of these two functions are shown in listing 4.1 where `f` is the applicative functor.

```
1 pure :: a -> f a
2 (<*>) :: f (a -> b) -> f a -> f b
```

Listing 4.1: Types of Applicative's minimal complete definition

The `pure` function is the equivalent of monad's `return`, it simply lifts a value into the applicative context. The other function `(<*>)` (which is typically pronounced "applied over" or just "apply"). Apply take in two arguments, both of which are applicative values. The first argument is function within an applicative context from types a to b, and the second argument is of type a. Apply returns a value of type b inside of the same functional context. Apply "extracts" the function from the first argument and the value from the second argument and applies it to the function, all within whatever the applicative context is.

4.1.1 Other useful functions

Though `pure` and apply are the only two functions that are required to be defined to declare an instance of applicative there are several other useful functions that can either be derived from these two functions or come from other typeclasses which will be briefly covered here. First there are two variations on apply.

```
1 (*>) :: f a -> f b -> f b
2 (<*) :: f a -> f b -> f a
```

These functions sequence actions and still perform the contextual effects of both of their arguments but discard the value of the first and second argument respectively. These functions are used when some operation affects the applicative context but their returned value will not affect the final result of the applicative expression. For example when writing parsers it is common to have to consume some characters from the input without those characters affecting the final result of the parser.

A consequence of the applicative laws is that every applicative's functor instance will satisfy the following (McBride and Paterson 2016):

```
1 f <$> x = pure f <*> x
```

The next section will cover how these functions can be used in an applicative style of programming.

4.2 The Applicative Programming Style

In McBride and Paterson (2008) the authors prove that any expression built from the applicative combinators can take the following canonical form:

```
pure f <*> is_1 <*> ... <*> is_n
```

Where some of the `is`'s have the form `pure s` for a pure function `s`. Due to the rule mentioned at the end of the previous section this canonical form can also be expressed using the infix version of fmap `(<$>)`.

```
f <$> is_1 <*> ... <*> is_n
```

This is the form that most programs will take when they are refactored from a monadic style.

Context-free parsing is a good use case of the applicative type and many examples in this chapter are taken from parsers defined using the parsec library (Leijen and Martini 2006). The first example of the applicative programming style is a function that parses money amounts of the form `<currency symbol><whole currency amount>.<decimal amount>` e.g. “$4.59" or “£64.56".

```
data Currency = Dollar
              | Pound
              | Euro

data Money = M Currency Integer Integer

parseMoney :: CharParser () Money
parseMoney = M <$> parseCurrency <*> readWhole <*> readDecimal
```

The `parseMoney` function is in the canonical form as defined by McBride and Paterson (2008). The pure function `M` is lifted into the `CharParser` context and its three arguments are provided by three smaller parsers that handle the currency symbol, the whole amount, and the decimal amount separately.

The only difference between `readWhole` and `readDecimal` is that `readDecimal` has to consume the decimal point before reading the number. Instead of duplicating that number code let's perform a small refactoring to lift the parsing of the decimal into the `parseMoney` function which will allow us to reuse the `readWhole` function.

```
1 parseMoney :: CharParser () Money
2 parseMoney = M <$> parseCurrency <*> readWhole <* char '.' <*>
      readWhole
```

Here we can see that the result of parsing the decimal point is discarded because of the use of `<*` rather than the full apply. All of the variations of apply are left associative so the following definition of `parseMoney` causes a type error.

```
1 parseMoney :: CharParser () Money
2 parseMoney = M <$> parseCurrency <*> readWhole <*> char '.' *>
      readWhole
```

This error can be corrected by wrapping "`char '.' *> readWhole`" in parenthesis.

The canonical style of applicative functions is not always the most idiomatic way to define things. The following function parses strings surrounded by double quotes.

```
1 parseStr :: CharParser () String
2 parseStr = char '"' *> (many1 (noneOf "\"")) <* char '"'
```

`parseStr` does not match the canonical form because no lifted pure function is applied to the rest of the applicative chain. This function could be transformed to canonical form by pre-pending "`id <$>`."

The examples covered in this section give a basic introduction to programming in an applicative style. The next section will discuss common applications that are particularly well suited to definition in the applicative style and can be transformed from the monadic style.

4.3 Applications of the Refactoring

There are two things that make a particular application a good candidate for this refactoring. First, and most obviously, the application must be able to be defined using the applicative interface. Finally a good candidate will already have a large corpus of code that is already written in the monadic style. If a particular library already encourages its users to use it using applicative functors rather than monads then there is little work for the refactoring to do.

4.3.1 Parsec

Parser combinator libraries such as parsec (Leijen and Martini 2006) provide a simple way of building parsers from predefined smaller parsers (a.k.a. the combinators). The applicative interface is sufficient for defining parsers of context-free languages[1]. Despite this much of the code written using Parsec is monadic. A good example of this comes from Real World Haskell (O'Sullivan, Goerzen and Stewart 2008).

```
1 csvFile :: GenParser Char st [[String]]
2 csvFile =
3   do result <- many line
4     eof
5     return result
```

This can be very simply rewritten using the applicative like so:

```
1 csvFile :: GenParser Char st [[String]]
2 csvFile = many line <* eof
```

Shorter code is not always better however, in this case, I would argue that the applicative style is easy to clearer. The parser can be read left to right many lines are followed by the end of file.

[1]This is mostly true. The applicative interface can parse context-sensitive languages if your grammar is an infinite size (Yorgey 2012).

4.3.2 Yesod

Another possible application of this refactoring applies to parts of code used to define Yesod webservers (Snoyman 2016). The preferred way to handle the creation and processing of web forms is using the applicative interface (Snoyman 2012). Yesod doesn't force forms to be handled applicatively because a monad instance is provided as well but it is the *idiomatic* way to handle web forms. This refactoring would allow people to write in a monadic style and then refactor their code to fit the standard.

4.3.3 Other applications

4.4 Refactoring Monadic Programs to Applicative

This section will cover the mechanics of refactoring monadic code to the applicative style. Many of these examples are taken from the parser for money amounts and a JSON parser. The full source of these parse can be found in Adams (2016b) and Adams (2016a).

Take for example the following parser that parses strings that begin and end with double quotes.

```
1 parseStr :: CharParser() String
2 parseStr = do
3   char '"'
4   str <- many1 (noneOf "\"")
5   char '"'
6   return str
```

This parser first consumes a double quote (`char '"'`) then parses at least one other character other than double quotes and assigns those characters to the variable named `str`[2], finally the closing quote is consumed and `str` is returned. This particular

[2]This line is composed of two parser combinators, `many1`, and `noneOf`. `many1` takes another

function can be rewritten in an applicative style like so:

```
1 parseStr :: CharParser() String
2 parseStr = char '"' *> (many1 (noneOf "\"")) <* char '"'
```

The refactoring goes through the monadic version of the function line by line and composes computations with the applicative operators. In this case the first line of the `do` block does not affect the final result so it is followed by the `*>` which performs the action on the left hand side of the operator but discards that value. The next line's value is assigned to the variable str which is returned by the function so this means that on both sides of this computation the operator that composes it with its neighbours will have to "point" to it as well[3]. To determine which operator needs to be used on the right of the call to `many1` the refactoring needs to look at the next line and see if it also contributes a value to the final result. If it does then it and `many1` will be composed by the `<*>` operator, but since `char` doesn't, the operator between the call to `many1` and the second call to `char` becomes `<*` which discards the value `char` returns.

This is a fairly simple function to convert to applicative style. Let's look at another example that adds in more complexity by having multiple computations that contribute to the final value of the function. This function comes from the money parser that was used in 4.2 as well.

---

parser as its argument and applies it one or more times returning a list of the results. `noneOf` takes in a list of characters and succeeds if the current character is not in the provided list. Then the character is returned.

[3]This means `<*>` could occur on both sides, `*>` on the left, or `<*` on the right of the computation

```
parseMoney :: CharParser () Money
parseMoney = do
  currency <- parseCurrency
  whole <- many1 digit
  decimal <- (option "0" (do {
                    char '.';
                    d <- many1 digit;
                    return d}))
  return $ M currency (read whole) (read decimal)
```

The `parseMoney` function parses text into the `Money` data type. It works by first consuming the currency symbol and getting the appropriate `Currency` type from that. Then the whole money amount is read from one or more digits. Finally an "option" parser attempts to match a decimal point followed by one or more digits. If that fails then the decimal amount is zero. These three values are then combined into type `Money` which is returned. `parseMoney` can be rewritten in an applicative style like so[4]:

```
parseMoney :: CharParser () Money
parseMoney = M <$> parseCurrency <*> readWhole <*> readDecimal
       where readWhole = read <$> many1 digit
             readDecimal = read <$> option "0" (do {
                    char '.';
                    d <- many1 digit;
                    return d}))
```

On the left hand side of the chain of applicative actions there is a call to a pure operation, in this case the constructor `M`. Any pure computations that “collect" the values returned from applicative computations will appear on the left of the chain of operations. The pure computations are composed with applicative computations with the

[4]The where clause in this example has been included for formatting and readability and would not be generated automatically.

<$> operator which lifts the computation into the applicative context. The three computations are composed together with the <*> operator because they all contribute to the final result of `parseMoney`.

The chains of applicative computations can get more complicated. The following snippet of code parses a single in a JSON object which consists of a string key and a value which can be any valid JSON value, separated by a colon, and stores the key and value in a tuple.

```
objEntry = (,) <$> (spaces *> parseStr <* spaces <* char ':'
   <* spaces) <*> (parseJVal <* spaces)
```

When there are a large amount of computations that do not affect the final value of the function as a whole there can be multiple valid ways the chain can be structured. The `objEntry` function can be defined in several different ways as shown below.

```
objEntry = (,) <$> (spaces *> parseStr <* spaces <* char ':')
   <*> (spaces *> parseJVal <* spaces)

objEntry = (,) <$> (spaces *> parseStr) <*> (spaces *> char
   ':' *> spaces *> parseJVal <* spaces)
```

Both of the above versions of `objEntry` are equivalent to the first version. The automated refactoring will produce the first version of `objEntry`. The refactoring in general will produce applicative chains according to the following rules. Both sides of the apply operator will be parenthesised statements. After the first value producing operation every side effect causing operation will be composed with `(<*)`. In general the produced applicative chain will take the following form.

```
f = pf <$> (is_1 *> ... is_n *> vs_1 <* js_1 ... <* js_n) <*>
   (vs_2 <* ks_1 ... <* ks_n) ... <*> (vs_n ...)
```

The `vs` are functions that return values to be passed to the pure function `pf` all other functions just run within the applicative context without affecting the returned value of

`f`.

4.5 Preconditions of the Refactoring (When is a Monad actually a Monad?)

Many functional refactorings have non-trivial preconditions that must hold before the refactoring can be applied (Thompson and Li 2013). Fortunately this refactoring only has a single fairly simple precondition, the function to be refactored must be definable with the applicative interface not just the monadic interface. What does this mean exactly? Where is the line between applicative and monadic? Let's start by looking at the type signatures of the bind and apply functions.

```
1 (<*>) :: Applicative f => f (a -> b) -> f a -> f b
2
3 (>>=) :: Monad m => m a -> (a -> m b) -> m b
```

One thing to keep in mind is that these two functions' arguments are in opposite order. The key difference becomes clearer when examining the type of apply when its arguments are flipped.

```
1 flip (<*>) :: Applicative f => f a -> f (a -> b) -> f b
```

The only difference between this version of apply and bind is in the second argument. Bind takes in a function that takes in a value of type `a` and returns an `m b` whereas apply receives an applicative functor that contains a function from `a` to `b`. This means that within a monadic context bind allows access to the pure value contained in the monad while all of the arguments to apply are contained within the applicative functor.

What does this mean in practice? For example, the following functions are taken from a StackOverflow answer by Conor McBride (McBride 2011).

```
1 iffy :: Applicative a => a Bool -> a x -> a x -> a x
2 iffy ab at af = pure cond <*> ab <*> at <*> af where
```

```
3   cond b t f = if b then t else f
4
5 miffy :: Monad m => m Bool -> m x -> m x -> m x
6 miffy mb mt mf = do
7   b <- mb
8   if b then mt else mf
```

Both of these functions attempt to implement an if statement, `iffy` does it with `pure` and `(<*>)` whereas `miffy` uses the monadic functions. Both functions' first argument contains a boolean within its computational context. Depending on the value of this boolean the second or third argument is then returned. Though both of these functions will return the same value when given the same arguments, the effects within the context will be very different. When `iffy` is run all of the contextual effects will be run regardless of which value is returned.

If a value retrieved from the monadic context is used in a right hand side expression before the return statement then that function cannot be refactored to use the applicative interface without changing the contextual effects of the function.

4.6 Variations and Related Refactorings

This section will discuss related refactorings and variations on the generalise monad to applicative refactoring that may be useful. Related refactorings help transform code so that it can pass the preconditions. In this case it may be possible to extract monadic code into another function making the top level function definable with the applicative interface. Variations on refactorings slightly change the behavior of the refactoring. This section will present two variations, one which will inline small do blocks of monadic code automatically, another will recursively refactor do blocks that may occur inside of higher level statements.

4.6.1 Extract monadic code

Say someone wanted to refactor the following code to use the applicative interface rather than the monadic one it currently uses.

```
1 f = do
2   x <- getX
3   b <- getB
4   y <- if b then getY1 else getY2
5   log y
6   return (x,y)
```

This code will not pass the precondition because both `b` and `y` are used on the right as well as the left hand side of the equation. However, lines three through five don't really affect the rest of the function so they could be refactored into their own function then f could be rewritten applicatively.

```
1 f = (,) <$> getX <*> g
2
3 g = do
4   b <- getB
5   y <- if b then getY1 else getY2
6   log y
7   return y
```

4.6.2 Inline do blocks

Instead of extracting an entire function as in subsection 4.6.1 a developer may prefer to just inline a do block. This is useful if the monadic section is fairly small.

```
1 f = do
2   x <- result1
3   y <- result2
```

```
  z <- result3
  log z
  return (x,y)
```

```
f = (,) <*> result1 <*> (result2 <* do{z <- result3; log z})
```

Normally the variable `z` would prevent the function from being refactored. Introducing the small do block allows for a simple readable applicative function to be produced.

It is worth noting that if the variable `z` was also included in the output of the function the do block inlining would still work with a slight modification.

```
f = (,) <*> result1 <*> result2 <*> do{z <- result3; log z;
   return z}
```

4.6.3 Refactor Inner `do` blocks

If we take another look at the `parseMoney` function and it's applicative counterpart the default behaviour of the refactoring preserves the inner do block passed to the `option` parser.

```
parseMoney :: CharParser () Money
parseMoney = do
  currency <- parseCurrency
  whole <- many1 digit
  decimal <- (option "0" (do {
                    char '.';
                    d <- many1 digit;
                    return d}))
```

```
9    return $ M currency (read whole) (read decimal)
10
11 parseMoney :: CharParser () Money
12 parseMoney = M <$> parseCurrency <*> readWhole <*> readDecimal
13         where readWhole = read <$> many1 digit
14               readDecimal = read <$> option "0" (do {
15                                  char '.';
16                                  d <- many1 digit;
17                                  return d}))
```

It is perfectly possible to refactor this inner do block to the applicative style as well.

```
1
2  parseMoney :: CharParser () Money
3  parseMoney = M <$> parseCurrency <*> readWhole <*> readDecimal
4          where readWhole = read <$> many1 digit
5                readDecimal = read <$> option "0" (char '.' *>
                     many1 digit)
```

Chapter 5

Introducing Effectful Abstractions

Up to this point the data refactorings that have been discussed are changing abstractions that already existed in the source code. This chapter will explore refactorings that introduce effectful abstractions into pure code. In particular this chapter will focus on introducing monads and applicatives into pure code.

The `Identity` monad is the monad the does not embody any computation strategy (Gill, Newbern and Palamarchuk 2016). This means that any pure Haskell function could be refactored to be within the Identity monad. This refactoring can take in a set of functions and produce a corresponding set of functions with a monadic type. Take for example this definition of the Fibonacci numbers.

```
1 fib :: Int -> [Int]
2 fib 0 = 0
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
```

This can be refactored to the Identity monad like so:

```
1 fib :: Int -> Identity [Int]
2 fib 0 = return 0
3 fib 1 = return 1
4 fib n = do
5   x <- fib (n-1)
6   y <- fib (n-2)
7   return (x + y)
```

The new code could then be rewritten very easily (just by changing the type signature) to be within another monad. This allows for a developer to quickly create programs that can take advantage of monadic features such as IO or state. The monadic code could also be generalised to use applicatives with the refactoring detailed in chapter 4 but I also hope to develop a more straightforward way to introduce applicatives for this chapter.

Finally another abstraction that would seem to have quite a bit of potential for automatic introduction is the `Arrow` typeclass. Arrows were originally introduced as a more general abstraction to monads in Hughes (2000). The full relationship between arrows, monads, and applicative functors was more fully described in Lindley, Wadler and Yallop (2011). Given the relationship between these three typeclasses I believe it will be worth exploring introducing arrows as well. A possible outline of this chapter would be:

1. Introducing the Applicative style
2. Automated Monadification
3. Introduction to Arrows
4. Discussion of Arrows, Applicative, and Monads
5. Refactoring to Arrows

Chapter 6

Related work

There are several bodies of literature that are related to my thesis work. Other functional refactoring tools such as Wrangler (Li et al. 2006) are of obvious interest. There is also the code smell tool for Haskell HLint (Mitchell 2014).

Another interesting project is the Type-and-Transform system developed at the University of Utrecht (Leather et al. 2012). Which is a system for performing semantics preserving type changing transformations for the simply typed lambda calculus, and the polymorphic lambda calculus.

In general this chapter will briefly discuss the research done in two primary areas:

1. Refactoring and code smell tools for functional programming languages
2. Type changing program transformations

Chapter 7

Conclusion

Summarise my contribution here. The main contributions of my thesis are the development of type changing refactorings for GHC. With a particular emphasis on changing the abstractions that programs use.

Bibliography

Adams, S. (2016a). GenApplicative. https://bitbucket.org/sadams601/lhsnotes/src/262bdc8a49a587af52963fb436e3a3d97b80792f/genApplicative.lhs?at=master&fileviewer=file-view-default, [Online; accessed 21-June-2016].

Adams, S. (2016b). MoneyParser. https://bitbucket.org/sadams601/lhsnotes/src/262bdc8a49a587af52963fb436e3a3d97b80792f/moneyParser.lhs?at=master&fileviewer=file-view-default, [Online; accessed 21-June-2016].

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley.

GHC (2015). 1.5. release notes for version 7.10.1. https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/release-7-10-1.html, [Online; accessed 20-June-2016].

Gill, A., Newbern, J. and Palamarchuk, A. (2016). Control.Monad.Identity. https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Identity.html, [Online; accessed 20-June-2016].

Hughes, J. (2000). Generalising Monads to Arrows. *Science of computer programming*, 37(1), pp. 67–111.

Hughes, R. (1986). A novel representation of lists and its application to the function "reverse". *Information processing letters*, 22(3), pp. 141–144.

Leather, S. et al. (2012). Type-and-transform systems. Tech. rep., Technical Report UU-CS-2012-004, Department of Information and Computing Sciences, Utrecht University.

Leijen, D. and Martini, P. (2006). Parsec: Monadic parser combinators. `https://hackage.haskell.org/package/parsec`, [Online; accessed 20-June-2016].

Li, H. et al. (2006). Refactoring erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden*.

Lindley, S., Wadler, P. and Yallop, J. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5), pp. 97–117.

McBride, C. (2011). Applicatives compose, monads don't. `http://stackoverflow.com/a/7042674/1245646`, [Online; accessed 21-June-2016].

McBride, C. and Paterson, R. (2008). Applicative Programming with Effects. *J Funct Program*, 18(1), pp. 1–13.

McBride, C. and Paterson, R. (2016). Control.Applicative. `https://hackage.haskell.org/package/base-4.9.0.0/docs/Control-Applicative.html`, [Online; accessed 20-June-2016].

Mitchell, N. (2014). Hlint. `http://community.haskell.org/~ndm/hlint/`, accessed: 2014-05-2014.

O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). *Real World Haskell*. O'Reilly Media, Inc., 1st edn.

Snoyman, M. (2012). *Developing web applications with Haskell and Yesod*. " O'Reilly Media, Inc.".

Snoyman, M. (2016). yesod-core. `https://www.stackage.org/package/yesod-core/`, [Online; accessed 21-June-2016].

Thompson, S. and Li, H. (2013). Refactoring tools for functional languages. *Journal of Functional Programming*, 23(03), pp. 293–350.

Yorgey, B. (2012). Parsing context-sensitive languages with Applicative. `https://byorgey.wordpress.com/2012/01/05/parsing-context-sensitive-languages-with-applicative/`, [Online; accessed 21-June-2016].