# 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

# 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 it's dependencies (in particular ghc-exactprint).

# 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.

# Chapter 4

# Generalising Monads to Applicative

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 the GHC as the typeclass `Applicative`. An interesting part of the history of the GHC is that despite McBride and Paterson proving in their original functional pearl that all monads are also applicative functors, however, the 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`.

```
class Functor f where
  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.

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.

```
pure :: a -> f a
(<*>) :: 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.

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

```
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.

```
parseMoney :: CharParser () Money
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.

```
parseMoney :: CharParser () Money
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.

```
parseStr :: CharParser () String
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 the

## 4.4 Refactoring Monadic Programs to Applicative

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

## 4.6 Variations and Related Refactorings

### 4.6.1 Inline do blocks

```
f = do
  x <- result1
  y <- result2
  z <- result3
  log z
  return (x,y)
```

$$\Longrightarrow$$

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

### 4.6.2 Reordering of monadic statements

```
g = do
  x <- getChar
  y <- getInt
  return $ f y x
```

# Chapter 5

# Introducing Monads

This chapter will cover the refactoring of pure code to become monadic. This chapter will be structured much like chapter 4 with sections covering the motivation behind the refactoring, examples of the refactoring, the preconditions that must hold before applying the refactoring, and variations and related refactorings.

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.

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

This can be refactored to the Identity monad like so:

```
fib :: Int -> Identity [Int]
fib 0 = return 0
fib 1 = return 1
fib n = do
```

```
  x <- fib (n-1)
  y <- fib (n-2)
  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.

# Chapter 6

# Related work

There are several bodies of literature that are related too 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.

# Chapter 7

# Conclusion

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

# Bibliography


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, R. (1986). A novel representation of lists and its application to the function ï£¡reverseï£¡. *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. 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.