TYPE-CHANGING REFACTORINGS IN HASKELL

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

Draft on June 17, 2016

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 is probably going to be written last.

Chapter 2

Refactoring Haskell in HaRe

History of HaRe and the update to the GHC. Maybe some architectural details.

Chapter 3

Data refactoring in a functional context

Introduce some of the ideas of functional data refactorings. Talk about simple refactoring examples like Hughes lists and Maybe generalising to 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

TODO: Write more of an introduction here about what exactly the applicative context is before going into the applicative operators.

To implement the applicative typeclass two functions must be defined, `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.

Another operator that is very useful for programming in the applicative style (discussed further in section 4.2) is the infix synonym of `fmap`.

```
1 (<$>) :: Functor f => (a -> b) -> f a -> f b
```

A consequence of the applicative laws is that every applicative's functor instance will satisfy the following (GHC 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) McBride and Paterson prove that any expression built from the applicative combinators can take the following canonical form:

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

TODO: Include my variation on the canonical form which composes f with the infix fmap operator instead?

Where some of the `is`'s have the form `pure s` for a pure function `s`. 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.

4.3 Applications of the Refactoring

TODO: Maybe move this to earlier in the chapter? Since the previous section uses parsers as examples which is a major use case this section may end up being fairly redundant. Perhaps this should just be a subsection before section 4.1

4.4 Refactoring Monadic Programs to Applicative

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

TODO: Main precondition is that no variable assigned in the do block can be used on the RHS before the return statement. There can also only be a single return statement, e.g. no branches in computation.

4.6 Variations and Related Refactorings

4.6.1 Inline do blocks

TODO: Some examples of monadic code are only monadic in small chunks so the whole function may be able to take on more of an applicative structure but with a small do block embedded in it. See example below

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

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

4.6.2 Reordering of monadic statements

TODO: The statements in an applicative chain need to be ordered in the way the pure constructor takes the arguments. For example if the function `f :: Int -> Char -> String` was the pure constructor in the following monadic statement:

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

TODO: The generalisation refactoring would produce `g = f <*> getChar <*> getInt` which throws a type error. Swapping lines two and three before attempting

the generalisation would fix this.

Chapter 5

Mysterious third chapter of contribution

More research goes here.

Chapter 6

Related work

Mention type and transform, Meng Wang's paper...

Chapter 7

Conclusion

Well its done..

Bibliography

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.

GHC (2016). Control.applicative. https://hackage.haskell.org/package/base-4.9.0.0/docs/Control-Applicative.html.

Leijen, D. and Martini, P. (2006). Parsec: Monadic parser combinators. https://hackage.haskell.org/package/parsec.

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.