DATA-DRIVEN REFACTORINGS FOR HASKELL

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Stephen Adams

October 2018

Abstract

Agile software development allows for software to evolve slowly over time. Decisions made during the early stages of a program's lifecycle often come with a cost in the form of technical debt. Technical debt is the concept that reworking a program that is implemented in a naive or "easy" way, is often more difficult than changing the behaviour of a more robust solution. Refactoring is one of the primary ways to reduce technical debt.

Refactoring is the process of changing the internal structure of a program without changing its external behaviour. The goal of performing refactorings is to increase code quality, maintainability, and extensibility of the source program. Performing refactorings manually is time consuming and error-prone. This makes automated refactoring tools very useful.

Haskell is a strongly typed, pure functional programming language. Haskell's rich type system allows for complex and powerful data models and abstractions. These abstractions and data models are an important part of Haskell programs. This thesis argues that these parts of a program accrue technical debt, and that refactoring is an important technique to reduce this type of technical debt.

Refactorings exist that tackle issues with a program's data model, however these refactorings are specific to the object-oriented programming paradigm. This thesis reports on work done to design and automate refactorings that help Haskell programmers develop and evolve these abstractions.

This work also discussed the current design and implementation of HaRe (the *Ha*skell

*Re*factorer). HaRe now supports the Glasgow Haskell Compiler's implementation of the Haskell 2010 standard and its extensions, and uses some of GHC's internal packages in its implementation.

Acknowledgements

First I would like to thank my supervisor Professor Simon Thompson, for his guidance and patience throughout this process. Without his help this thesis would not have been possible.

I also need to thank my parents, Jeff and Diane, for their support and encouragement. You both have been so excited for me to take this opportunity. It can't have been easy to have me live an ocean away, and your support in me started this project has been amazing.

A special thanks go out to Kristin Lamberty, Elena Machkasova, and Nic McPhee the professors of computer science at the University of Minnesota, Morris. All three of you introduced me to the world of computing and I wouldn't be here today without all of your help. You prepared me as much as anyone can be prepared for a PhD.

Alan Zimmerman is cited many times in this thesis but I don't think that is sufficient thanks for all the work he has done. Without you HaRe would not be where it is today. Your dedication to the Haskell open source community is much appreciated.

I would like to extend my thanks to Adriana and Gaya Perera for letting me stay with them for the final year of my degree. You very graciously opened your house to me when I needed it and allowed me to stay much longer than any of us were expecting.

Finally I must thank Rosemary for her love and support. You have supported me through some of the toughest times of my life. Without you this would not have been finished, and without you this accomplishment would mean nothing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Functional Programming

Functional programming is a programming paradigm that focuses on data values as described by expressions which are built from function applications and definitions (Reade 1989). Functions in this case are closely related to the idea of mathematical functions. What qualifies a programming language as functional is debatable but several concepts are often included in languages that are described as functional such as immutability and first class functions.

An immutable language is one where once some value has been created it cannot be modified. First class functions mean that functions are allowed to be treated like any other data type. First class functions can be passed to or returned from from other "higher-order" functions. Iteration is accomplished through recursion rather than via looping. There is also a heavy emphasis on functions remaining "pure," that is without side effects; though functional languages do provide ways to use IO or state, it is emphasised that functions should remain pure if at all possible. Haskell's type system allows for effectful computations to be contained within monadic types.

```
1 f x y = case x > 0 of
2   True -> x - 1
3   False -> x + 1
```

Figure 1: A simple function

1.2 Haskell

Haskell is a statically typed, lazily evaluated, pure functional language. Haskell also supports Hindley-Milner type inference (Hindley (1969); Milner (1978)). Type inference means that Haskell programs do not need the type of every top level binding to be explicitly provided by the programmer. Types will be *inferred* at compilation time so that every part of a Haskell program's type is known at that time. Haskell's type system also allows users to define and use their own types.

Lazy evaluation, also known as call-by-need (Wadsworth 1971), means that Haskell expressions are not evaluated when they are passed as a parameter to a function, but rather when that value is used. For example, in the Haskell function `f`, in Figure 1, the parameter y will never be evaluated. Lazy evaluation is more nuanced than call-by-name style parameter passing. Composite data types will be evaluated only as much as required to allow the computation to continue. This allows for both partial and infinite data types, for example `[1..]` is the list containing all of the natural numbers.

Haskell is also a pure language. Purity is the idea that functions cannot perform actions in addition to returning values. These additional actions are known as side-effects. Haskell allows for traditionally side-effect causing operations (IO, state, etc.) through the use of monads. Monads represent computations, which when run will have effects as well as producing a result. A principle topic of this thesis is refactoring to support patterns of computation related to Monads and other related structures.

```
1 data Choice = Yes | No
```

Figure 2: A simple data type that models a yes/no choice.

```
1 data Choice = Yes String | No
```

Figure 3: `Yes` `Choice`s now contain a string as well.

1.2.1 User defined types in Haskell

All high level programming languages come with some predefined set of types and, typically, some way for users to define new types. In Haskell new types can be introduced with the `data` keyword. Figure 2 shows a simple data type, `Choice`. There are only two different values a `Choice` can be, `Yes` or `No`, these are the *constructors* of `Choice`.

Data types can also extend existing Haskell types. Returning to the `Choice` example maybe we want to be able to store some text along with a `Yes` to explain why this choice was made. This modified definition of `Choice` is shown in Figure 3. Now a call to the `Yes` constructor needs to be passed some value of type `String` to construct a value of type `Choice`.

Haskell allows you to take this one step further however. The creator of a type does not need to specify exactly which types will be stored within the new type, this means that types can be *parameterized*. Figure 4 shows a version of `Choice` where the `Yes` choices can contain any type `a`. This is known as parametric polymorphism (HaskellWiki 2015b). This type of polymorphism allows users to effectively reuse their types again and again without having to redefine them every time a new "inner" type is needed.

```haskell
data Choice a = Yes a | No
```

Figure 4: `Yes Choices` can now hold any other type.

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance (Eq a) => Eq (Choice a) where
  Yes x == Yes y = x == y
  No == No = True
  _ == _ = False

  c1 /= c2 = not (c1 == c2)
```

Figure 5: The `Eq` type class and `Choice`'s implementation of it.

1.2.2 Type classes

Now that we have seen how users of Haskell can define their own data types, and how types can be defined in a generic way, making the definitions applicable to a whole set of arguments. Haskell is a functional language so a natural next question is to wonder that, if we can define types that are generic in some ways, can the same be said of functions? The answer is yes, this is what type classes do for Haskell.

A type class allows a user to define a set of functions and/or constant names and their types. Then definitions of these functions and constants can be given for specific types. Figure 5 shows the type class for `Eq` which describes what it means for types to have a notion of equality. Figure 5 also shows how the definition of `Choice` from Figure 4 implements the `Eq` type class. The "`(Eq a) =>`" clause in the instance declaration is a constraint on the type of `a`. This means that for a `Choice` to be "`Eq`-able" `a` must be as well.

Haskell gives its users very powerful tools to help construct their own types and define those types' properties. These tools allow for many powerful concepts to be defined abstractly making them reusable. One of the more well known abstract concepts

of the Haskell language are monads which are the subject of the next section.

1.2.3 Monads

Monads are a type class used to represent "composable computation descriptions" (HaskellWiki 2015a). Haskell uses monads to augment pure computations with features that other languages would allow as side effects such as state or I/O. Haskell has a built in monad type class, whose declaration is shown in Figure 6. Instances of monad need to implement two functions: `return` which will create a monadic computation that will produce its parameter and »= (which is pronounced bind). Bind's first argument is some monadic value with a result of type `a` and its second argument is a function that takes in a value of type `a` and returns another monadic value that computes a result of type `b`. Bind computes the result of type `a` from the monadic value it was given in its first argument and passes that value to the function it received as its second argument. This description of bind only applies when its first argument is capable of computing some result or when the monad instance does not need to perform any additional tasks. This is the great power of monads as an abstraction, because bind can be implemented in almost any way[1] it can act as a way to specify how a particular set of computations are performed.

The `Choice` type that was defined in the previous section is really just a copy of the `Maybe` type which is commonly used to represent computations that may fail or return a result. Instead of `Yes` and `No` `Maybe`'s constructors are `Just` and `Nothing` respectively. `Maybe` is a monad and in its case bind is used to cause a chain of computations to return `Nothing` if one of the steps of the computation fail. This allows for functions that could potentially fail to return a result to be composed without having to check if their parameter is a `null` value. Figure 7 shows this in practice. Each of the `f_` functions can be written with the assumption that their parameter exists, only the top level function, `g`, needs to check if the `chain` function has failed.

[1] All instances of monad should abide by the three monad laws (Wadler 1995)

```
class Applicative m => Monad m where
return :: a -> m a
(>>=) :: forall a b. m a -> (a -> m b) -> m b
```

Figure 6: The `Monad` type class declaration

```
chain :: a -> Maybe a
chain a = f_1 a >>= f_2 >>= f_3 >>= f_4

g a = case (chain a) of
  (Just b) -> putStrLn ("Got value: " ++ (show b))
  Nothing -> putStrLn "Something went wrong"
```

Figure 7: A chain of functions composed using bind. Each `f_` function is of type `a -> Maybe a`

Monads are a powerful abstraction but it is also useful to further categorise monads that exhibit different properties. Haskell comes with type classes, that inherit from `Monad`, and that categorise additional features monads can have, such as failure (`MonadFail`) or failure and recovery (`MonadPlus`) (Yorgey 2009b). `MonadPlus` is a type class that represents the type of computations that may fail and also provide some way of choosing between possibly failed computations.

In addition to the type classes that further categorise monads, monads themselves inherit from other more general types: functors and applicative functors. Functors are the set of types that can mapped over, in the case of lists the values can be transformed to another value by mapping some function over the list. Applicative functors are the set of types that can have sequential actions performed over them. Applicative functors will be discussed in more detail in Chapter 6.

The Haskell community has built a large system of types around monad for representing all different sorts of computations. This system of types provides a fertile environment for data-driven refactorings, the topic of this thesis.

1.3 Refactoring

Refactoring is the process of changing a program without changing its behaviour. This is done to improve its internal structure (Fowler 1999). The term refactoring was first coined in (Griswold 1992) and these ideas expanded to the object-oriented paradigm in (Opdyke 1992). Martin Fowler's 1999 book "Refactoring" has become the canonical reference for object-oriented refactorings (Fowler 1999). People have been performing refactoring for much longer and it was called "program restructuring" in the literature (Kuck (1981); Ferrari and Lau (1976)).

Behaviour preservation is what separates refactoring from other types of program manipulation. This idea of behaviour preservation is that refactoring will not introduce new bugs or eliminate old ones. To prevent semantic changes after refactoring, many refactorings have non-trivial preconditions (Mens, Demeyer and Janssens 2002). For example, the refactoring that renames an item (such as a function or variable) should check that the new name being introduced does not cause a name clash with a name already used in the source program.

Manual refactoring can be tedious and error prone because changes to small portions of code may require system-wide changes. When deleting a parameter from a function, for example, every call site of that function also needs to be modified, and missing even a single call site will cause an error. Refactoring by hand depends on high testing coverage to ensure that functionality is preserved (Fowler 1999). This means that tools that can automatically perform refactorings and ensure that preconditions are met are highly desirable.

1.3.1 Refactoring Functional Languages

Refactoring a functional language has a few key differences from refactoring an imperative language. The higher-order nature of functional languages means that any subexpression of a function is a candidate for generalisation whereas in other languages

the types of parameters and results are limited. This means that functional refactorings can target much more of the source program's abstract syntax tree. The semantics of functional languages also allow for more comprehensive checking of preconditions based on the static semantics of the language (Thompson and Li 2013).

It is also not unusual for functional refactorings to be substantially different than their object-oriented (OO) counterparts. For example creating a `case` statement from a multi-equation function definition in a functional language versus inlining virtual methods into a `case` statement in an OO language require substantially different program manipulations (Li 2006). The first refactoring, the inlining of a multi-equation function, involves moving the pattern matches from each of the target function's equations to the left hand side patterns of the new `case` expression. The object oriented refactoring needs to look up all of the implementations of the virtual method, inline their bodies into the new target method, and build a case statement that switches depending on which concrete type the new method is called invoked in. Both of these refactorings are consolidating some conditional logic into a single function,[2] but the syntax elements that need to traversed and constructed to build the target program are very different. Additionally there can be refactorings with no OO counterpart, monadification the introduction of monadic types into otherwise pure code for instance.

1.4 Summary

One of the most widely accepted best practices in software development is the concept of incremental change, an essential concept of both the extreme programming and agile software development philosophies (Beck (2000); Fowler and Highsmith (2001)). Refactoring remains a key step in this incremental development process to preventing technical debt from causing development to slow to a crawl.

Much of the refactoring literature focuses on changes that need to be made to the

[2] If we consider methods to be functions that happen to be associated with a particular object.

structure of programs. The structure of a program is very important and structural refactorings can maintain the "separation of concerns" by extracting functions from existing definitions or ensuring that a program's names reflect what the program currently does. This thesis argues that it is just as important to maintain the ways that a program structures and evaluates the data it computes.

This thesis has chosen to use the term "data-driven" to describe the type of refactorings that are prompted by the data that a program computes. These refactorings can be prompted by an insufficiently fine-grained data model. For example, the "introduce type synonym" refactoring which creates a new type synonym that helps separate certain instances of a type that are used to represent different concepts.

This thesis also takes advantage of Haskell's call-by-need evaluation strategy which allows for control flow to be abstracted by the user. The "generalise monad to applicative refactoring" (see Chapter 6) takes code that was formally monadic and sequentially evaluated and makes it possible to evaluate it in parallel. The "generalise maybe" refactorings from Section 4.4 takes a concrete effect and makes it an abstract one that can be instantiated in multiple ways. Both of these refactorings are applied as either the programmer's understanding of the data they are working with becomes more nuanced or to prepare the program's data model for enhancement. In this way the data "drives" these refactorings.

1.5 Thesis Outline

This thesis will proceed as follows:

Chapter 2: Related work

This thesis begins with chapter that discusses the some of the related work that this thesis builds from and is inspired by.

This chapter covers other refactoring tools for other functional and object-oriented languages. This chapter also discusses multiple projects that are using refactoring in

unique and interesting ways such as introducing parallelism, and non-traditional programming languages. Next the chapter describes work on developing syntactic sugar for applicative functors. Finally there is a brief discussion of type changing program transformation systems.

Chapter 3: Background: Refactoring Haskell in HaRe

This chapter covers the history of HaRe (the **Ha**skell **Re**factorer), the technologies it depends on, and the current implementation of HaRe. Specifically there is an overview of the (GHC API 2016), the generic traversal library Scrap Your Boilerplate (Lämmel and Jones 2003), and ghc-exactprint (Zimmerman and Pickering 2016). This chapter also describes how the inner workings of HaRe are implemented and the functions that compose its API. Finally it discuses the design and development process of implementing refactorings for HaRe.

Chapter 4: Data-Driven Refactorings

Chapter 4 introduces data-driven refactorings. The chapter begins with a discussion of this type of refactoring for object-oriented languages. The rest of the chapter describes data-driven refactorings for the Haskell programming language. First there is a description of the "introduce a type synonym," the "renaming" refactoring of data-driven refactorings. The renaming refactoring is the most straightforward refactoring, it simply changes the name of a variable to one that better suits the real meaning of the variable. Similarly a type synonym can rename types to better reflect what certain instances of that type are being used for in a program, and this refactoring supports that transformation. The other two refactorings covered in this chapter are the "generalising maybe" and "list to Hughes list" refactorings.

The "generalising maybe" refactoring (Section 4.4) rewrites functions that use the concrete type of `Maybe` to use, instead, the operations provided by the type classes it implements, `Monad` and/or `MonadPlus`. The final refactoring described by this chapter is the "list to Hughes list" refactoring (Section 4.5). Hughes lists (also known as difference lists) are an alternative implementation for lists that support $O(n)$ time

appends. This refactoring takes functions that use that standard list implementation and rewrites the function to use Hughes lists instead. The approach for this refactoring is applicable between any two types that are "reversibly embeddable" a concept that will be defined as well.

Chapter 5: Implementing Data-Driven Refactorings in HaRe

This chapter continues from the refactoring designs presented in chapter 4 to describe HaRe's implementation of both the "maybe to MonadPlus" and the "list to Hughes List" refactorings. There is also be a discussion of the API that supports the "list to Hughes List" refactoring and can be used to define further "reversibly embeddable" type refactorings. Finally this chapter will describe the enhancements made to HaRe's API, specifically the addition of high-level transformation functions.

Chapter 6: Generalising Monadic code to Applicative Functors

This chapter presents another generalisation refactoring. Applicative functors are a, relatively, new addition to the Haskell environment. Applicative functors are an interface for sequencing effectful computations. Currently the Haskell community predominantly uses the monadic interface for effects. This chapter will describe the design and implementation of a refactoring for taking a monadic `do` statement and transforming it to use the `Applicative` (the Haskell type class for Applicative functors) interface instead.

This chapter also describes how the Haskell community is currently using the `Applicative` interface, based on the results of a survey of Hackage, the Haskell package archive (GHC 2016). Finally this chapter concludes with a discussion of possible applications of the refactoring.

Chapter 7: Introducing Monads

Chapter 7 describes the monadification refactoring. Monadification is the process of introducing monads into pure code. Monads are the standard way for effects to be used in Haskell and so a refactoring to automatically add them is very useful to the community. There are many styles of monadification and this chapter describes several

of them. Finally it discusses the implementation of the monadification refactoring in HaRe.

Chapter 8: Conclusion

The final chapter summarises the work done for this thesis and the contributions it has made. There is also a discussion of the current Haskell tooling ecosystem, the challenges currently facing Haskell tool builders, and how they could be tackled. It concludes with a discussion of future work that could be performed on HaRe.

1.6 Contributions of this Research

The work in this thesis was carried out in HaRe. This study focused on adding additional refactorings to HaRe of a new type. Rather than being motivated by the structural problems of a program, data-driven refactorings seek to resolve issues that are caused by the data types a program uses. The contributions of this research are:

- Extending the HaRe API to better support data-driven refactorings. These refactorings are complex and require more information from the abstract syntax of GHC than prior refactorings in HaRe. The contributions to the API were focused on analysis of the types of nodes and creating a higher level interface for common small expression level transformations. These changes are described in Chapter 5.

- The design and implementation of the "generalise maybe" and "list to Hughes list" refactoring, described in Chapters 4 and 5. The "generalise maybe" refactoring is a way of taking a concrete effect and turning it into an abstract one so that it can be instantiated in multiple ways. The "list to Hughes list" refactoring describes a way to rewrite a program's data model to use a different type with a similar interface.

- The design and implementation of the "generalise monads to applicative" refactoring, Chapter 6. Effects in Haskell are typically handled using monads, a powerful abstraction that allows the type system to check the type of the effects that are being performed by a program. Since the introduction of monads the Haskell community has developed more fine-grained approaches to effect handling. This refactoring allows software systems to handle effects using the applicative as opposed to the monadic interface.

- The implementation of the "monadification" refactoring, see Chapter 7. Monadification is the process of making a program work over a monadic type rather than a pure one. This refactoring supports the transformation of pure programs to effectful ones.

Chapter 2

Related Work

This chapter reviews current work in refactoring and other areas of the literature and helps to situate the work described in this thesis into a broader landscape. This chapter begins, in Section 2.1, with a short description of refactoring tools, including the sophisticated modern IDEs that support object-oriented languages and the refactoring tools that support functional languages.

Next Sections 2.3 and 2.4 will describe some of the work done to transform programs so that they are executed in parallel rather than sequentially. The original idea behind refactoring was to reduce the technical debt of a target program. Technical debt is the idea that when a system is being implemented there can be quick and messy ways to implement things, but this implementation will incur "debt" on the project. Much like financial debt some debt is needed to get a project off the ground but also like financial debt if debt is allowed to grow unchecked it can grind a project to a halt (Cunningham 1992). Refactoring, was traditionally the process of "paying back" this debt, however the potential scope of refactorings has been expanded to include, for example, introducing parallelism rather than just improving code quality.

Section 2.5 covers work done in the program transformation and refactoring fields. In particular earlier versions of monadification are discussed, such as "Reuse by Program Transformation" (Lämmel 1999), and "Monadification of Functional Programs"

(Erwig and Ren 2004). Other work that is covered in this section includes work on refactoring and program transformation that is particularly focused on types.

The final section of this chapter discusses how refactoring tools are implemented. This section pays special attention to refactoring tools that target languages other than Haskell, because the focus of Chapter 3 is on building refactoring tools for Haskell.

2.1 Refactoring Tools in Modern IDEs

Refactoring tools have become a standard feature in integrated development environments. The four most popular IDEs for object-oriented languages, Eclipse[1], NetBeans[2], IntelliJ[3], and Visual Studio[4] all come with refactoring tools for their primary language (Burazin 2014). These refactoring tools support some general refactorings (renaming, method extraction) and some that are specific to object-oriented languages (pushing/pulling methods up/down the object hierarchy).

Modern IDEs, however, were not designed with functional programming in mind. Eclipse, Netbeans, and IntelliJ were all built to support Java and Visual Studio supports multiple languages most prominently C++ and C#.[5] These tools allow community developed plugins to expand the languages they can support but the plugins can fall out of date, for example the Haskell Eclipse plugin was stopped being supported in 2015.[6] Though modern IDEs are heavily depended on in object-oriented development functional programming languages have developed their own, mostly independent tooling ecosystems. This will be the focus of the next section, refactoring tools for functional programming languages.

[1] `https://www.eclipse.org/`
[2] `https://netbeans.org/`
[3] `https://www.jetbrains.com/idea/`
[4] `https://www.visualstudio.com/`
[5] Visual Studio also includes support for F# out of the box, but this could be considered the exception that proves the rule.
[6] See: `https://wiki.haskell.org/IDEs#EclipseFP_plugin_for_Eclipse_IDE`

2.2 Refactoring Tools for functional languages

A reason that often used to be given to explain why functional languages are not in widespread use in industry is the lack of a robust tooling ecosystem (Wadler 1999). This is no longer the case as functional language ecosystems have undergone a great deal of development in recent years and, maybe coincidently, use in industry has gone up substantially in the last five years. This section will cover some existing refactoring and code smell tools for functional programming languages.

2.2.1 HLint

HLint is a "code smell" tool for Haskell. Poorly designed code often produces "smells," apparently superficial problems that indicate deeper design issues (Fowler 1999). One of the most common of these smells is duplicated code. Other hints include simplifying boolean expressions to remove unneeded calls to `not` (e.g. "`not (a == b)`" should become "`(a /= b)`"), or replacing common types of folds with their prelude defined names (e.g. `sum` can replace `foldr (+) 0`). A code smell tool suggests changes to a code base such as alternative functions to use, how to simplify code, and redundancies (Mitchell 2014).

Code smell and refactoring tools are very closely related. Simplistically a code smell tool detects problems in a code base and a refactoring tool fixes them. If a tool can detect a problem why can't the same program fix them? HLint has a `-refactor` flag that will automatically apply the suggestions. However there can be a problem in doing this, e.g. a single piece of code could have multiple smells, how would HLint choose which one to apply? Also once a transformation has been applied other hints may no longer be applicable. HLint's behaviour in these cases is not documented.

One of the powerful features of HLint is its customizability. An HLint configuration file (called `hlint.yaml`) added to the root of a project will be detected by HLint and it will suggest both the default hints as well as the custom hints from that file. Hints are

```
- hint: {lhs: x !! 0, rhs: head x}
```

Figure 8: A simple hint from (Mitchell 2014)

very simple to write.

Figure 8 contains the definition of a hint that detects the list index operator is being used to look up the 0th element of a list and suggests using `head` instead. The `lhs` tag is the code HLint will search for. If code matching `lhs` is found HLint will suggest the code be replaced with the `rhs` code. HLint assumes any single character variable is a substitution parameter. Given the hint from Figure 8 and the following code:

```
f list = list !! 0
```

HLint produces the following output.

```
 Suggestion: Use head
Found:
  list !! 0
Why not:
  head list

1 hint
```

A major limitation of HLint is that it is only aware of a single module at a time. HLint is not aware of what types or names are in scope and code smells that span multiple modules cannot be discovered.

2.2.2 Haskell Tools Refact

HaRe is not the only refactoring tool for Haskell. In late 2016 Haskell Tools Refact was announced and is currently at version 0.7 (Haskell-tools 2017a). The Haskell Tools project is a GHC based developer tool kit for writing transformations (Haskell-tools

2017b). There are eight refactorings currently supported[7].

- Rename
- Generate type signature
- Generate exports
- Extract binding
- Inline binding
- Organize imports
- Float out
- Organize extensions

Haskell Tools has implemented its own abstract syntax tree. The AST of Haskell Tools is generated using information from all of GHC's compiler stages. Each node represents the same language elements; it just includes additional information that is spread across the different stages of the GHC (Haskell-tools 2017a).

The Haskell Tools refactorer is currently integrated into the Atom editor[8] with Sublime Text[9] support planned for the near future (Haskell-tools 2017a)

2.2.3 Wrangler

Wrangler is a refactoring and code inspection tool for Erlang (Li et al. 2006). Erlang is a functional programming language designed to be massively scalable and highly fault tolerant (erl 2015). It was originally developed in 1986 by Joe Armstrong, Robert Virding, and Mike Williams at the Computer Science Laboratory at Ericsson Telecom AB (Armstrong 2007). Erlang's core design tenets include lightweight processes, that

[7] May 2018
[8] `https://atom.io/`
[9] `https://www.sublimetext.com`

```
1 ?T("erlang:spawn(Arg@)")
2
3 ?T("erlang:spawn(Arg@@)")
```

Figure 9: Some Wrangler templates

communicate through message passing. Erlang also boasts a “let it fail" error handling architecture, when a process fails the error is not handled by the process where it occurred, but is handled by a separate dedicated part of the program (Armstrong 2003).

Wrangler is accessible from the command line and has been integrated into both Emacs and Eclipse. It currently supports a large library of refactorings, code smells, as well as other program analysis tools such as clone detection and automatic API migration (Li et al. 2006). Additionally Wrangler supports a template-based API and a domain specific language which allow users to define their own refactorings and script composite refactorings (Li and Thompson 2011).

The template-based API of Wrangler allows users to define program analyses and transformations using Erlang concrete syntax. Wrangler templates consist of fragments of Erlang syntax that may contain meta-variables or meta-atoms that can stand for any language element. Meta-variables/atoms are variables or atoms that end with the "@" character; this meta-variable/atom binds a language element to its name so that that element can be referred to by name in the definition of the refactoring. Meta-variables/atoms that end with "@@" are list meta-variables/atoms that match a sequence of language elements as long as they are of same sort (Li and Thompson 2012).[10]

The first template in Figure 9 matches applications of `erlang:spawn` when it is called with one argument whereas the second template will match the same function with any number of arguments.

Composite refactorings are refactorings that are made up of multiple refactorings

[10]Things like the arguments to a function or a sequence of expressions in a function body are the same "sort."

run, in sequence, one after the other. It can be challenging to develop composite refactorings if they are not explicitly handled by the refactoring tool. The naive solution just chains refactorings together with the output from one refactoring in a composite refactoring becoming the input to the next refactoring. However, what if the second refactoring fails in a chain of four? Composite refactoring definitions, without tool support, become filled with error handling code to manage the situation when one of the component refactorings fail. Wrangler defines a domain specific language that helps describe the various facets of a composite refactoring.

The Wrangler DSL supports the creation of a composite refactoring through a variety of features. First, Wrangler extends every primitive refactoring with a *refactoring command generator*. A command generator allows the extended refactoring to accept not just concrete values but also structures that specify how the parameter should be generated; each parameter of a command generator accepts either a concrete value, a condition that checks if a value is satisfactory, or a generator for creating the parameter based on the previous parameters. A refactoring for renaming functions named with the format `camelCase` to `camel_case` would accept three arguments: the target filename, the name of the target function, and the desired new name. This command generator's first parameter is a condition that always returns true because any file is a valid target for renaming. The second parameter is another condition that checks if the function name matches is in camel case format (e.g. "`aFunName`"). The final parameter is generated by taking the second parameter and modifying it so that the name is in the corresponding "snake case" format (e.g. "`a_fun_name`").

The DSL also allows decision making to occur during the execution of a composite refactoring. Composite refactorings are transactional and can be either atomic or non-atomic. Atomic composite refactorings require each component refactoring to be successfully applied before continuing onto the next refactoring. If a single refactoring fails inside of an atomic composite refactoring, the entire refactoring fails and the

program remains unchanged. When a single refactoring fails inside a non-atomic composite refactoring, correspondingly, the entire refactoring will not fail and continue by trying the next refactoring in the sequence. The Wrangler DSL allows for refactorings to described as atomic and non-atomic sections at each level.

2.2.4 RefactorErl

Another notable tool for the Erlang language is RefactorErl. RefactorErl started out as another refactoring tool for Erlang but has since expanded into a source code analysis and transformation tool (Horpácsi 2013). RefactorErl uses a "semantic program graph" to represent an Erlang source program, this graph is broken up into three layers (Tóth and Bozó 2012):

1. Lexical layer: This is where token, spacing, and comment information is kept about the source program.

2. Syntactic layer: This layer keeps the abstract syntax tree of the source program.

3. Semantic layer: This contains additional calculated semantic information about the source program, such as module and function references as well as variable bindings.

Refactoring tools, in general, only require the information contained within these first two layers, the semantic layer helps RefactorErl implement its static analysis capabilities. The semantic program graph is constructed after a source program's abstract syntax has been obtained from the Erlang language front end. The semantic layer is built on top of the abstract syntax tree by several different static analysers that each add a different kind of information to the graph. For example, the function analyser adds a semantic function node when the first reference to or the definition of a function is found. Every reference to that function discovered after that points to the original node (Tóth and Bozó 2012).

The edges in the semantic layer are also labeled so the relationships between nodes can be captured as well. The function analyser, for instance, labels the edges going from the semantic function node to internal references to that function with a *funIref* label, and the *fundef* label connects the semantic node to the function definition it represents. The semantic layer is critical for the static analysis RefactorErl performs but it is also quite useful for the refactorings as well. Consider the "move a function" refactoring[11], which moves a function definition from one module to another, every reference in the target function needs to be checked to ensure that those definitions are available in the new scope of the function. Without a semantic layer a refactoring tool may have to traverse the syntax tree multiple times to locate all the references the target function contains and ensure they are available in the function's new location. With the semantic layer, on the other hand, finding all of the references can be done in two steps over the graph (one step from the reference to the semantic node, the next step following the semantic node's "definition" edge, e.g. *fundef*, *fielddef*, etc.).

RefactorErl currently supports 24 different refactorings as well as a suite of static analysis and program comprehension tools.

2.2.5 ROTOR

A newcomer to the refactoring tools for functional language space is ROTOR[12] the first refactoring tool to target OCaml (Rowe and Thompson 2017). Language features of OCaml provide some unique challenges for a refactoring tool. In OCaml one module may be included in another so that, for example, when renaming the function `f` in module `A` but `A` is included in module `B` then both `A.f` and `B.f` will need to be renamed. ROTOR handles this by decomposing refactorings into a set of textual replacement operations that can depend on other transformations (Rowe and Thompson 2017). From

[11] See: `http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/MoveFunction`

[12] **R**eliable **O**Caml-base **T**ool for **O**Caml **R**efactoring

```
type Strategy a = a -> Eval a
```

Figure 10: The strategy type

the renaming example renaming "`A.f`" would depend on the "`B.f`" renaming succeeding and vice versa.

An opportunity of the ROTOR project is that it has an partner in industry, Jane Street Capital. ROTOR is using the core library[13] an "industrial strength" version of the OCaml standard library as a test bed for testing the refactoring tool.

2.3 Refactoring to introduce parallelism

The reasons to refactor source code have also expanded beyond code quality. This section will describe two different projects that have developed refactorings to change the execution of a program from single to multi-threaded. Functional programming languages are well suited to parallel execution due to immutability by default and in some languages (such as Erlang) first-class concurrency features. This section will first describe the "ParaForming" which uses refactoring to introduce parallel abstractions into Haskell code, then it will describe work done to refactor Erlang code to introduce algorithmic skeletons.

2.3.1 ParaForming

ParaForming is an approach to construct parallel programs from an existing program using software refactoring (Brown, Loidl and Hammond 2011). The ParaForming work targets Glasgow parallel Haskell: "GpH", an extension to Haskell, and is implemented in HaRe. Parallelism is added to programs in GpH using strategies (see Figure 10).

A strategy takes its argument and determines how it will be evaluated inside of the `Eval` monad. The `rpar` strategy introduces parallelism by "sparking" its argument.

[13] `https://github.com/janestreet/core`

```
1 sumEulerSeq :: Int -> Int
2 sumEulerSeq n = sum (map euler (mkList n))
```

Figure 11: A sequential calculation that sums the Euler totient function

```
1 sumEulerPar1 :: Int -> Int
2 sumEulerPar1 n = sum (map euler (mkList n) `using` parList
    rdeepseq)
```

Figure 12: A refactored version of the function from Figure 11

Sparks are tasks that are collected into a pool which is managed by the runtime. The spark pool is a source of work that GHC can pull from when there are idle processors. Sparks may be evaluated in parallel or not at all depending on the availability of spare cores.

The simplest parallel refactoring is to introduce data parallelism. This refactoring is applied to an expression that works over a list and evaluates each member of that list in a spark. A sequential function that sums the Euler totient function is in Figure 11 and the refactored program is in Figure 12

This refactoring evaluates the calculation of `map euler (mkList n)` using[14] the `parList rdeepseq` strategy. The `parlist`[15] function evaluates each element of a list in parallel according to a given strategy and `rdeepseq` is the strategy the fully evaluates its argument.

The refactored program in Figure 12 is highly parallel but not very efficient because the parallelism is too fine grained. Another refactoring can help in this case instead of sparking every element of a list another strategy can be introduced, one that separates the list into "chunks" and each of the chunks of the list is executed in parallel. This refactoring adds an additional argument to the function that determines how many chunks the list will be split into, as seen in Figure 13.

[14]The (`using :: a -> Strategy a -> a`) function just evaluates some expression with the given strategy.

[15]`parlist :: Strategy a -> Strategy [a]`

```
1 sumEulerChunk :: Int -> Int -> Int
2 sumEulerChunk c n = sum (map euler (mkList n) `using`
    parListChunk c rdeepseq)
```

Figure 13: A "chunked" version of the function from Figure 11

These two refactorings are both a way of introducing data parallelism with varying degrees of granularity. The other form of parallelism is known as task parallelism. Where data parallelism is focused on computing different parts of a data structure in parallel (the elements of a list in the previous case), task parallelism instead focuses on having different “tasks” excecuted in parallel. The work done in Brown, Loidl and Hammond (2011) outlines a refactoring that can make recursive calls happen in parallel.

2.3.2 Cost-Directed Parallel Refactoring

The previous section touched on one of the big challenges of parallel programming, determining the correct level of parallelism to achieve maximum performance. Brown et al. (2014) describes a methodology to introduce algorithmic skeletons into Erlang programs using the Erlang refactoring tool Wrangler. In addition to introducing a skeleton this work provides cost models that estimate the performance of the program after adding each skeleton. This estimate helps a programmer to make an informed decision about which parallelisation strategy is the best for a particular program.

An algorithmic skeleton is a common parallel pattern. A skeleton is implemented as a higher-order function that takes in a sequential function and any parameters that the skeleton requires. Brown et al. (2014) discusses the four most common and useful skeletons. For example, the map skeleton works by breaking up the target data into pieces that can be operated on in parallel. Finally the results from the the parallel computations are combined back into a single image. One of the examples presented in Brown et al. (2014) is an image processing system that denoises images. Denoising a section of an image can be done independently from processing the other sections of

the same image. The introducing the map skeleton would break the image into pieces to be denoised in parallel then the outputted sections can be stitched back together again

Skeletons are simple to understand in theory but it can be difficult to know which to apply in practice. This is when the cost models of each skeleton become useful to help make an informed decision about which skeleton should cause the greatest speed up and this information can be used to guide the refactoring. In Brown et al. (2014) an initial benchmark of the program can be used to estimate the speed up that different skeletons could provide.

Parallelisation can be tedious and difficult to do which makes it a good candidate for tool assistance. A refactoring tool can guide a programmer through the process of parallelisation. Much like how the data-driven refactorings have multiple small changes are required before the entire process can be considered "finished" changing a program to run in parallel is also a sequence of several smaller changes.

2.4 ApplicativeDo

Haskell is a pure functional programming language. Purity in this context means that Haskell is side-effect free. This causes some confusion because in most other languages side-effects are allowed, if not the primary way that programs produce their "result." Haskell instead models side-effect causing computations using Monads, which take effects of computation that are typically implicit and make them an explicit result of the program instead. Beyond modeling side effect causing operations, monads allow for computations to be supplemented with additional features (HaskellWiki 2015a).

Haskell supports the writing of monadic code through the "`do`" syntax sugar, an example of this syntax can be seen in Figure 14. Monads and their `do` syntax have become a commonly accepted pattern for handling effects within the Haskell community.

Applicative functors are another type class that describe computations performed within some context, but are less powerful than monads. They were first described

```
1 f = do
2   x1 <- A
3   x2 <- B x1
4   x3 <- C
5   return (x2,x3)
```

Figure 14: A simple monadic function constructed using a `do` statement.

by McBride and Paterson (2008), and a fairly recent change to GHC made the `Applicative` type class[16] a superclass of `Monad` which means that every instance of `Monad` now must also implement the `Applicative` interface as well.

Compiler changes can't force a community to change its practices and `Applicative` remains under-utilised compared to `Monad`. This under-utilisation of applicative functors in Haskell has not gone unnoticed; in Marlow et al. (2016), an implementation of a language extension for GHC was introduced that changes the way Haskell interprets `do` statements so that applicative functors can be supported by the same `do` syntactic sugar. `Applicative`'s offer a key advantage over monads, when applicative functors are composed the results they calculate remain independent of each other. This means that values under applicative functors can be evaluated in parallel. Using the familiar `do` syntactic sugar to also support `Applicative`'s means that programs can become concurrent "for free," this is the main motivation behind the applicative-do work (Marlow et al. 2016).

Figure 14 shows a simple function constructed using Haskell's do notation. The standard way that this function would be desugared is shown in Figure 15. Finally Figure 16 shows how the same function would be desugared when the `ApplicativeDo` language extension is turned on.

The *ApplicativeDo* algorithm will attempt to insert as many applies into the expression as possible. When the implementation of the `Applicative` instance evaluates the two arguments of apply in parallel, better performance can be achieved by adding

[16] `Applicative` is what the GHC calls the type class that implements applicative functors.

```
f = A >>=
  (\x1 -> B x1 >>=
    (\x2 -> C >>=
      (\x3 -> return (x2,x3))))
```

Figure 15: The desugared version of `f` from Figure 14.

```
f = (\x2 x3 -> (x2, x3))
       <$> (A >>= (\x1 -> B x1))
       <*> C
```

Figure 16: How `f` will be desugared when applicative do is turned on.

more applies.

There could be multiple ways to desugar a particular function. The *applicativeDo* algorithm first assumes that every expression has an identical time cost, and from this assumption the algorithm heuristically determines the desugaring with the shortest execution time.

2.5 Program transformations

Refactoring is a type of program transformation but it does not constitute the whole field. A major difference between refactoring and other types of program transformations is that a refactoring must take into account the human readability of its output. Program transformations typically focus on just the algorithm whereas refactorings must take into account the broader effects a transformation has on a codebase and the context that programs exist in. Additionally the target program of a refactoring needs to be readable, maintainable, and keep proper layout and user comments. Other types of program transformation don't typically have these concerns. This section will describe some of the program transformation work most relevant to this thesis. First it will describe the type and transform system developed by (Leather et al. 2015). Next there will be a discussion of the previous methods of monadification found in the literature.

```
rep :: A -> R
abs :: R -> A

rep . abs = id
abs . rep = id
```

Figure 17: The properties that must hold for the type-and-transform system to work over types `A` and `R`

2.5.1 Type and transform systems

The type-and-transform system described in (Leather et al. 2015) is a system for a semantics preserving and type changing program transformations over the typed lambda calculus with let polymorphism. The type-and-transform system is limited to isomorphic types, there must be a way to convert between the two types and back again as described in Figure 17.

The type-and-transform system supports type-changing rewrites through typed rewrite rules that insert conversions between the source and target types as appropriate. To handle the fact that there are multiple ways to retype a program each rewrite rule is weighted to maximize the use of the target type, introduce the target type as soon as possible in the program, and delay the conversion back to the source type as late as possible.

This work emphasises formalisation and its correctness and the work is done in the context of the lambda calculus rather than a full programming language. There is a Haskell implementation of their system but it is only a prototype though they state that they want to expand this work to work with Haskell however this has not been published yet.

```
1 f (g x) (h y)
2
3 let x1 = g x in
4   let x2 = h y in
5     f x1 x2
```

Figure 18: A-normal form converstion

2.5.2 Automatic Monadification

Monadification is not a new problem and various solutions have been presented in the literature. In (Lämmel 1999) monadification is performed in two steps. First the program is transformed into A-normal form[17], which flattens applications into let expressions. The first line of Figure 18 shows a normal expression and line 3 of the same Figure shows that expression in A-normal form.

Once the program has been converted into A-normal form, a let expression of the form:

$letx = t1 int2$

Is transformed into:

$t1 >>= \lambda x.t2$

If the right hand side of the lambda is not already a monadic type then `return` will be introduced, e.g. $t1 >>= \lambda x.return\ t2$. The full transformation is given by inference rules in (Lämmel 1999).

Monadification is developed further by (Erwig and Ren 2004). This work provides an algorithm for restricted call-by-value monadification as opposed to the semantics style inference rules defined in (Lämmel 1999). This work targets the lambda calculus extended with case and let expressions. The algorithm from (Erwig and Ren 2004) is very similar to the one implemented in HaRe. It has the same precondition that every call to a monadified function must be fully saturated, which means that every call site of a target function needs all of its parameters to be named variables, and it produces

[17] This is also known as sequencing

the same style of monadification as the implementation provided in HaRe. A prototype implementation of this method was produced as a part of (Erwig and Ren 2004).

The idea of providing refactoring as a monadification was first discussed in Reinke et al. (2005). This source provides a comprehensive discussion of the different styles of monadification. There are several different, equally "correct," ways to monadify a function. A refactoring tool, perhaps more than other types of program transformations, needs to consider what its users intentions and desires are for applying the transformation so that the most useful output can be produced. This is particularly important for refactoring tools because its output needs to be directly usable by developers. This makes the discussion of what monadification style to be very relevant since the monadified functions need to be easily usable. This issue of style is continued in Chapter 7.

2.5.3 Data Type Transformations

Data types are obviously the focus of data-driven refactorings and a considerable amount of work has been focused on them in the program transformation and refactoring literature.

In Kort and Lämmel (2003) the authors present a set of transformation primitives that help modify data types, either through writing scripts or via a GUI in an interactive mode. This framework can be used to define refactorings but its creators implemented operators that can be used for behaviour changing transformations as well, which take the framework's capabilities beyond refactoring. This framework, for example, supports the insertion and deletion of constructor components (Kort and Lämmel 2003).

This framework created a set of transformation operators that work over Haskell but the design of the framework is general enough to be implemented in other langauges that also support algebraic data types.

2.5.4 Generic Refactorings

The previous subsection discussed a transformation system for data types and one of the main contributions of that framework is its language agnostic design (Kort and Lämmel 2003). This section describes another language agnostic system for program transformation which was reported in "Towards Generic Refactoring" (Lämmel 2002). This work asks the question what types of refactorings can be applied to many different programming languages and even markup languages such as XML.

This paper provides an implementation of the refactoring framework in Haskell but also provides an interface that can be instantiated for many different languages. This is possible because, despite syntax differences between different programming languages, certain refactoring actions have the same purpose regardless of the target language though the refactorings will need to be pointed towards the correct syntax elements for the target language. A good example of this is the "extraction" refactoring. It's better known as "extract a function" or "extract a method" depending if the target language is object-oriented or functional. In any language the refactoring will introduce a name for a previously anonymous section of code. The syntax that the refactoring targets and what the extracted piece of code is, is what makes up the differences between languages. In Haskell the refactoring would target expressions and the extracted code is a function, in Java statements would be made into a new method (Lämmel 2002).

The canonical reference for refactorings, Fowler (1999), is written using Java and though the author stresses that the refactoring catalogue is useful to other languages, the descriptions of the refactorings are very specific to object-oriented programming. Lämmel (2002) helps identify that certain transformations can be applied regardless of languages. This thesis will expand on this question in Section 4.1 by examining the data driven refactoring chapters from Fowler (1999) that do not obvious functional equivalents.

2.6 Engineering refactoring tools

Sections 2.1 and 2.2 briefly discussed some refactoring tools and the features they provide but did not say much about how they are built. This section will look at how refactoring tools are actually built, with a specific focus on how these tools gain access to a representation of the target source code and how they work with that representation. This section will also focus on languages other than Haskell. Refactoring Haskell is, of course, the focus of this thesis, and the general implementation of HaRe will be the focus of Chapter 3.

This section will look in greater detail at implementing refactoring tools for Erlang and Clojure.

2.6.1 Implementing Wrangler

Wrangler was first mentioned in Section 2.2.3, here we describe in a bit more detail how Wrangler is implemented, with a particular focus on the internal representation of the Erlang source code that Wrangler uses and how that representation is transformed.

Erlang comes with its own "front end." A language front end provides access to some of the same tools that language compiler and runtime systems are built out of. Refactoring tools are mostly interested in a language's lexer, parser, the data type the language is represented as internally (this usually is an abstract syntax tree), and a pretty printer for that data type. Many languages do not store all of the information a refactoring tool requires in its internal representation and Erlang is no different. Wrangler is dependent on the SyntaxTools library which includes more semantic data about the target program (Li et al. 2008). In addition to the information SyntaxTools adds to the AST of an Erlang program the library also makes it possible to add additional information to the tree as well (Carlsson 2015). Wrangler uses this feature to further annotate the AST with both syntactic and semantic information (Li et al. 2008).

Once a refactoring tool has parsed the target program into its internal representation the transformations can begin. However the internal representation of a program for even moderately sized programs can be quite sizeable. Traversing and modifying these structures can introduce large amounts of what's called "boilerplate" code into a project. In this case, the boilerplate is highly repetitive code that simply walks through a structure, it is difficult to maintain and hides the relatively small amount of "real" code that is actually performing the transformations (Lämmel and Jones 2003). Generic programming is a technique used to eliminate this type of code. Rather than use a third-party generic programming library like HaRe does (see Section 3.3) Wrangler has implemented its own versions of the generic traversals it requires.[18] This is feasible to in Erlang because its weak dynamic type system means that traversals can be defined in a single function whereas Haskell needs separate functions defined to handle each type of the abstract syntax tree.

2.6.2 Refactoring LISPs

The LISP family of languages make interesting target languages for refactorings. Lisp's support for macros and because Lisp code is structured as lists means that the internal representation of a Lisp program can be manipulated with the list-processing functionality that comes built into the language.

Another unique aspect to building refactoring tools for a Lisp is the support that the Emacs text editor can provide. Implementations of the Emacs text editor normally ship with their own dialect of Lisp referred to as Emacs Lisp (Free Software Foundation 2015). Refactoring tools are commonly integrated into Emacs, for example both HaRe and Wrangler provide Emacs extensions in Emacs Lisp. In those cases all the Lisp code is used for is to define the the user interface and wrapper around the command line calls to the refactoring tool. If the target language of the refactoring is a Lisp then Emacs

[18] See: `https://github.com/RefactoringTools/wrangler/blob/master/src/api_ast_traverse.erl`

Lisp can be used to define refactorings directly. Consider the Clojure refactoring tool "clj-refactor.el" (clojure emacs 2018). Clojure is a Lisp that can be compiled to either Java bytecode for execution on the JVM, or to Javascript for front end web programming (Hickey 2018). The Clojure refactoring tool uses a separate analyser to generate an AST independent of which dialect is being targeted. Once that representation is created Emacs Lisp can actually be used to perform the source code transformation instead of Clojure.

2.7 Summary

This chapter has discussed the literature that relates to and helped build the work that is described in the rest of this thesis. It began with a discussion of other refactoring tools, for both functional and object-oriented languages. Next in Section 2.3 was a description of work that uses refactoring to introduce parallelism into programs. Refactoring's original goal was to improve code quality and reduce technical debt, but with this work refactoring has been expanded to include introducing parallelism. This thesis again pushes the scope that refactorings can focus on towards data type evolution. Section 2.4 describes an alternative interpretation for the Haskell `do` syntax. The work described in Chapter 6 has the same goal as this work, the introduction of applicative rather than monadic operations. The approach taken in Chapter 6 uses refactoring to explicitely introduce applicative operations rather than having the compiler do this implicitely. The program transformation field also has looked at how data types can change and some of this work was described in Section 2.5.

The final section discussed how refactoring tools were implemented in Erlang and Clojure. Both Wrangler and clj-refactor deal with similar issues when it comes to getting an abstract syntax tree of their target programs. Third party libraries help both of these projects gain access to the abstract syntax tree of the target program with enough information in them to perform transformations while preserving formatting. Other

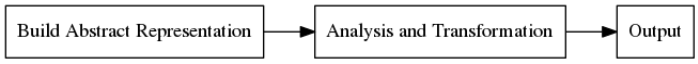


Figure 19: A general toolchain for program transformation.

Building Abstract Representation	Analysis and Transformation	Outputing Results
Parser	Generic Programming Library	Pretty-printer
Lexer	Custom Analysis	Token-Stream
Renamer	Traversal Schemes	IDE Functionality
Typechecker	IDE Functionality	
IDE Functionality		

Table 1: The components used during the different phases of program transformation.

refactoring tools perform much of the analysis required to perform transformations themselves such as RefactorErl and haskell-tools.

By considering how tools are built we can begin to understand the general toolchain that must be built to implement a program transformation tool. In general program transformation tools work in three main phases, Figure 19 shows the general toolchain that a tool will need to build. Table 1 shows the types of libraries and technologies that are used in each of the phases.

Before any transformation can happen the source program must be read into a representation that the tool can work with. Many tools use the existing compiler toolchain's lexer and parser to build an abstract syntax tree, this is what HaRe and Wrangler do. Additional compiler components, such as the renamer or typechecker, can be used to decorate the syntax tree with additional information. If a tool is implemented in a specific IDE, the IDE may provide functionality to work with the target language. The notable example of this would be clj-refactor which is built into Emacs, since Emacs is supports its own dialect of Lisp and Lisps place a strong emphasis on metaprogramming clj-refactor uses built in Emacs functions to move through and transform Clojure source files (clojure emacs 2018). Emacs uniquely supports Lisp but other IDEs also provide ways to interact with the abstract structure of the source code files being edited. Intellij IDEA, for example, provides "`PSIFile`" a generic interface for representing the

contents of a file as a hierarchy (JetBrains 2016). For a source code file the `PSIFile` would represent the abstract syntax of that language, but this model also allows for other types of languages, such as XML, to be modeled with the same interface.

The second phase of program transformation is the actual analysis and transformation of the target program's abstract representation. This phase is where the program works with and modifies the structure that was constructed during the first phase. In complex statically typed languages such as Haskell third party generic programming libraries are required to efficiently traverse and transform the abstract syntax of the source language. Other tools may perform additional analysis of their own and add that information to the abstract program structure. RefactorErl does this with the semantic graph that is built on top of the syntax tree. Wrangler also does this by augmenting the syntax tree with additional information after the standard Erlang parser has finished. Finally, an IDE may contribute its own analysis and transformation functionality, once again Emacs' support for Emacs Lisp gives the editor powerful tools for transforming other Lisps. The powerful object oriented focused IDEs offer basic rewriting capabilities but also powerful plugin environments so tools can depend on analysis and transformation tools built by community members.

The final phase a program transformation tool needs to perform is outputting the modified source. If the output needs to be seen by the user this process is handled by a "pretty-printer." Pretty printers takes some abstract representation of source code, typically an abstract syntax tree, and turns it into the code literal that it represents. Pretty printing can prove quite challenging depending on the abstract structure the tool is using. Parsers often omit some of the elements of a source file, code comments and whitespace are often omitted from abstract syntax because they aren't needed by the compiler. Tools that want to preserve these aspects of a user's program need to either store the formatting and comments in the abstract representation somehow, or reconstruct them directly from the token stream. Finally, an IDE can provide powerful

abstractions that can perform the pretty printing automatically. Intellij will automatically display changes made to a section of a PSI file, effectively solving this issue for tool builders that target Intellij.

The implementation of program transformation tooling, in general, follows the three stage pipeline of:

1. Constructing an abstract representation of the source program.

2. Analysis and transformation of that representation.

3. Outputting the transformed source code.

The design of these three phases are highly dependent on the source language the tool targets. In constructing the first phase of a transformation tool it is common to reuse lexer and parser of the target language's runtime if possible which makes the following phases dependent on the abstract syntax that is used byt the runtime system. What information is contained in the abstract syntax informs the steps that a tool must take in the next phase. If information that is required for the transformation isn't included in the abstract syntax it will need to be computed at this point. Additionally depending on the complexity of the abstract syntax tree a generic programming library may need to be introduced or traversal and transform schemes implemented to work with the abstract syntax. The final phase of a program transformation tool, printing, is again determined by the information that is available from the abstract representation. The original format of the source file may have to be stored separately if the abstract representation does not contain enough information to preserve the user's comments and formatting.

This generic transformation tool pipeline is how many program transformation tools are constructed and HaRe is no exception. This chapter has described other transformation and refactoring tools. The remainder of this thesis will focus on the Haskell refactoring tool HaRe. The next chapter will begin by describing how HaRe is constructed and how refactorings are implemented in it.

Chapter 3

Background: Refactoring Haskell in HaRe

Work on HaRe was initiated at the University of Kent. HaRe was started by Huiqing Li, Claus Reinke, and Simon Thompson in early 2003 (Thompson et al. 2010). The first implementation of HaRe supported the Haskell 98 standard. This first implementation of HaRe and it's first catalogue of refactorings were some of the main contributions of Huiqing Li's PhD thesis (Li 2006). Chris Brown continued to expand on HaRe with his PhD thesis "Tool Support for Refactoring Haskell Programs" (Brown 2008). In it he expanded the number of refactorings HaRe supports, HaRe's API, and built new code analysis tools, such as duplicate code elimination and program slicing transformations, using the infrastructure of HaRe.

The Haskell ecosystem has evolved a great deal since then. Haskell 2010 is now the formal language standard but the Glasgow Haskell Compiler (GHC) has become a de facto Haskell standard (Thompson and Li 2013). GHC supports the entire 2010 standard but also includes language extensions that can do everything from changing the type system to adding new syntax features (HaskellWiki 2017b).

Chris Brown began the working on updating HaRe so that it could support all of GHC Haskell and not just Haskell 98. Since then development of HaRe has been lead

by Alan Zimmerman; much of HaRe's current infrastructure is his work (Tools 2017). He also contributes to several of HaRe's dependencies including GHC-Mod (Gröber 2017), the current implementation of Strafunski-StrategyLib (Koppel and Alan 2018), and he started ghc-exactprint (Zimmerman and Pickering 2016).

The contributions of this thesis are built on top of all of the work that has gone into making HaRe what it is today. This chapter will describe the implementation of HaRe beginning, in Section 3.1, with a brief discussion of the original implementation of HaRe, and its dependencies such as Programatica (Hallgren 2003a), a language frontend for Haskell 98, and Strafunski-StrategyLib (Lämmel and Visser 2002) a generic programming library. Next the chapter describes the current dependencies of HaRe beginning with Section 3.2. This section describes the GHC API, the language frontend that is included with the GHC. HaRe now uses the GHC API for access to the internal representation of Haskell rather than a third-party library like Programmtica. In addition to Strafunski-StrategyLib HaRe's newer code also uses another generic programming library, *Scrap Your Boilerplate* (SYB) (Lämmel and Jones 2003). Section 3.3 describes how HaRe uses SYB to perform abstract syntax tree traversals and transformations. Section 3.4 describes, *ghc-exactprint*, the pretty-printer HaRe uses to output the modified program (Zimmerman and Pickering 2016). Next the chapter concludes with a description of the current implementation of HaRe in Section 3.5, including a description of HaRe's API, how refactorings are currently implemented, and how HaRe makes use of the GHC API. For the purposes of this thesis the "current" implementation of HaRe is, as of this writing, version 0.8.4.1[1].

3.1 The original implementation of HaRe

Implementing an automated refactoring system has several dependencies, including a frontend for the language that is targeted, a generic programming library or some

[1] `https://github.com/RefactoringTools/HaRe/tree/4055ef45f0de3c966fd7841986ab0ed2ee814055`

means of making generic traversals of abstract syntax trees, and a pretty printer. The language frontend is required for the refactorer to analyse and modify source code, the generic programming library assists in traversing the complex abstract syntax tree of a real-world programming language, and the pretty printer outputs as source the modified AST in a form recognizable by the author of the original program. The original implementation of HaRe fulfilled these dependencies with two libraries, Programatica, which provided the language frontend and pretty printer, and Strafunski-StrategyLib a generic programming library.

3.1.1 Programatica and Strafunski

Programatica was a project at the OGI School of Science and Engineering to build tool support for validating Haskell programs (Hallgren 2003b). The Programatica team open sourced their frontend so that other tools could also use it (Thompson et al. 2010). At the time there was no API to access the internals of GHC, so the HaRe team chose to use Programatica over other available front ends because it was the simplest front end that supported the full Haskell 98 standard along with a number of its extensions (Li 2006). Programatica's Haskell front end is broken up into multiple components including a lexer, a parser, an abstract syntax tree data type, a module system, a type checker, and a pretty printer. Programatica's frontend allows for the implementers of HaRe to focus on refactoring only rather than having to build all of these components as well.

Programatica's abstract syntax contains 20 data types with 110 data constructors in total. Working with the abstract syntax tree directly would introduce a large amount of "boilerplate" code into HaRe that would make maintenance and reusability more difficult (Li 2006). Instead HaRe used Strafunski-StrategyLib, a combinator library for generic programming, to traverse the abstract syntax tree (AST) of the source code (Lämmel and Visser 2002).

Commonly refactoring a program only modifies small sections of the source program's AST. Renaming a function, for example, will only need to modify the name

used in the binding and places where that name is being used; all other sections of the AST remain unmodified. A commonly used operation in Strafunski takes a function that works on a particular data type (or types)[2] and extends it to work on all types by leaving all other types unmodified. Strafunski also provides "strategies" that define how that extended function will be applied to the syntax tree as a traversal of the tree (Lämmel and Visser 2002)[3]. These two mechanisms, extending a function to work over all types and combinators that define how functions will be applied to a tree of values, are common to many generic programming libraries. A more specific example of using generic programming libraries is shown in Section 3.3.

These two dependencies allowed the original implementation of HaRe to obtain the abstract syntax tree of source code to be refactored, traverse and transform that AST, and output the modified program. All these tasks are things that the current implementation of HaRe needs to do but the dependencies that it relies on have changed somewhat. The Haskell standard was updated in 2010 (Marlow et al. 2010) and GHC continues to expand the number of language extensions it supports. Unfortunately Programatica has not kept up with these changes and does not support anything beyond Haskell 98. At the same time GHC's own API has been defined and matured so that it can replace Programatica as HaRe's front end. HaRe now relies on this and a few other projects for its language frontend. For generic traversals "Scrap Your Boilerplate" (Lämmel and Jones 2003), has been added as a dependency though Strafunski-StrategyLib is still used in some of the older parts of HaRe's code base and when certain traversal schemes that Strafunski provides are needed.

3.1.2 HaRe's original refactorings

HaRe's original refactorings fall into three categories, structural, module, and data-oriented refactorings.

[2] In the renaming example this could be the operation that checks if a variable is the one being renamed and replaces it with the new name.

[3] E.g. full top-down or full bottom-up.

3.1.2.1 Structural Refactorings

Structural refactorings principally concern the name and scope of entities defined in a program (Li 2006). These refactorings target functions, and smaller sections of code. A traditional example of this is the renaming refactoring. Renaming is the most basic refactoring. The purpose of renaming is to change the name of a given entity. A renaming refactoring could target a variable, function name, type, or any other entity that a programmer can assign a name to. This refactoring allows for the names used in a program to truly reflect what that program is actually doing.

Though these refactorings target small pieces of code, the changes made to the code base can affect modules throughout the codebase. The renaming refactoring, for example, can affect any other module that uses the renamed object.

Other examples of structural refactorings include Deleting a Definition, Duplicating a Function, and Adding an Argument (Li 2006, p. 15).

3.1.2.2 Module Refactorings

Module refactorings concern the imports and exports of an individual module, or the relocation of definitions between modules (Li 2006, p. 20). A simple refactoring in this category would be "clean an import list," which analyses a module's import list and removes redundant import declarations. Another example of such a refactoring would be the "move a definition" refactoring. As its name suggests, move a definition takes a definition from one module and moves it to another and fixes the imports and exports of any affected modules. For example if we were moving some function `foo` from module `A` to module `B` any external dependencies of `foo` needed to be imported into `B` and if those dependencies are no longer used in `A` then the relevant import statements should be removed. If `foo` is still being used in `A` then `A` needs to import `B` (if it does not already do so). Finally any other modules that currently depend on `foo` need to now import `B` (if it's not already imported) and remove the import of `A` (if none of `A`'s other definitions are used).

3.1.2.3 Data Type Based Refactorings

The third category of refactorings are those that are associated with data type definitions (Li 2006, p. 21). "Add field names" is a good example of a data-oriented refactoring. The add field names refactoring will add field names to a data type. These names can then be used as selector functions that make extraction of a particular part of a type a simple function call. The new field names are generated by HaRe but can be renamed by the user (Li 2006).

These original refactorings were chosen to be basic yet still useful, and for their ability to give insight into the issues surrounding implementing an automated refactoring tool (Li 2006). In addition to the refactorings that were implemented by HaRe's developers, an API was exposed so that other developers could implement their own refactorings this is discussed in more detail in Section 3.1.3.

3.1.3 The HaRe API

Early in the development cycle of HaRe, it was restructured to expose an API for implementing refactorings and general Haskell program transformations (Li, Thompson and Reinke 2005). The HaRe API contains a collection of functions for program analysis and transformation of Haskell 98 programs. These functions, along with the functionality provided by Strafunski and Programatica, form the basis for implementing basic refactorings (Li, Thompson and Reinke 2005).

The HaRe API exposes the full Programatica abstract syntax for Haskell 98 to the user but because of generic programming with Strafunski only the to be transformed parts of the AST have to be explicitly referenced in a refactoring (Li, Thompson and Reinke 2005). Another key feature of the API was to hide layout and comment preservation allowing the programmer to focus on program transformation instead. Each subtree in the AST is tagged with its absolute location in the source file. Any modifications to the AST will change the location of all elements that occur after the change.

The API abstracts over this cascade of changes that follows even the simplest of modifications. The HaRe API transformation functions modify the token stream and the AST simultaneously which keeps refactoring definitions free of this location bookkeeping (Li, Thompson and Reinke 2005). The token stream of a program is produced by the lexer, and is a list of all of that program's tokens in the order that they appear. The complete format of a program is only stored in the token stream but the full structure of a program is only clear when working with the abstract syntax tree. This means that refactoring's want to reason about the AST of source program but, to preserve formatting, must make sure that all modifications made to the program are also reflected in the token stream.

The overall goal of the HaRe API is to help ensure the correctness of new refactorings by limiting the amount of code required that is not related to program transformation, and isolating common error sources caused by having to keep the AST and token stream synced (Li, Thompson and Reinke 2005). This design goal still guides the development of HaRe, and many of the functions that were provided in the original HaRe API have been ported to HaRe's latest implementation.

As was mentioned in Section 3.1.1, HaRe's dependencies have changed somewhat in its current implementation. HaRe originally used Programatica as its front end, now the language front end is composed of several projects, the GHC API (GHC API 2016), ghc-mod (Gröber 2017), and ghc-exactprint (Zimmerman and Pickering 2016). The following sections will describe these dependencies as well as Scrap Your Boilerplate (Lämmel and Jones 2003), another generic programming library, and how HaRe currently uses them.

3.2 The GHC API

Rather than being a monolithic executable, the Glasgow Haskell Compiler (GHC) is composed of several smaller components that each correspond to a separate compiler

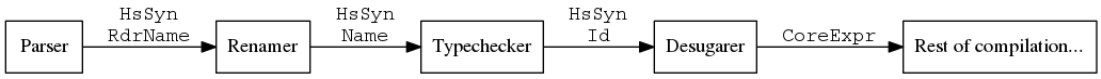


Figure 20: GHC Compiler stages.

stage. GHC's executable consists of a lightweight main function that ties together the smaller components (Marlow and Peyton Jones 2012). These components are exposed to users and this is what constitutes the GHC API.

3.2.1 Compiler stages of GHC

Some of the major components and the order in which they are used in GHC are shown in Figure 20 which has been adapted from Marlow and Peyton Jones (2012). This Figure is not a complete description of all of GHC's components, just the parts relevant to HaRe. The full diagram can be found in (Marlow and Peyton Jones 2012). The label after each compiler stage indicates the type of AST that is produced by that stage.

The top level data type for all of the GHC abstract syntax is `HsSyn` (Marlow and Peyton Jones 2012). `HsSyn` is parameterised by some identifier type; each compiler stage produces a different type of identifier with the additional information that stage produces. For example the typechecker takes in an AST parameterised by `Name` and returns an AST parameterised by `Id` which is a `Name` with additional type information.

3.2.2 GHC's Name Types

There are five name types that GHC uses, they are:

- `OccName` is the simplest type of name. It is just a wrapper around a `FastString`, which is an array of bytes, and an optional `NameSpace`. An `OccName` is contained in each of the other four identifier types. A `NameSpace` is an simple descriptor of what type of name the `OccName` comes from such as value, type constructor, or data constructor.

- `RdrName` names are produced by GHC's parser. `RdrName`s are essentially just an `OccName` with optional module information if the source name has been qualified.

- `Name`s are produced by the renamer, the component that resolves every name's scoping and binding. A `Name` contains an `OccName` and a `Unique` that differentiates `Name`s that have the same `OccName`. A `Unique` is just an integer but it is generated by GHC in such a way that it is guaranteed to be unique. They have also had their scoping and binding resolved, so each `Name` contains the source span where that name was defined and a data type that describes what sort of `Name` it is (e.g. a name defined internally in the module or a name from another module).

- `Id` & `Var` are the identifiers produced by the typechecker.[4] These identifiers contain a `Name`, and a `Unique`. Finally these names also contain the type of the variable they represent.

These identifier types change throughout the compilation process but these names only parameterise the syntax tree, the shape of the tree itself stays the same throughout compilation.

3.2.3 GHC's syntax tree

GHC's abstract syntax is currently made up of over 90 data types. Many of those types have multiple constructors; the expression data type `HsExpr` for example has over 40 constructors. This section will briefly introduce the structure of GHC's abstract syntax tree.

[4] `Id` is just a synonym for `Var`.

```
1
2 type Located e = GenLocated SrcSpan e
```

Figure 21: The `located` type.

```
1 type LHsExpr id = Located (HsExpr id)
```

Figure 22: The located expression

3.2.3.1 Common syntax types

The most common type in any given piece of GHC abstract syntax would be a `Located` as seen in Figure 21.

`Located` is used to tag syntax elements with their start and end positions in a source file. The `SrcSpan` contains the filename from which the span comes and then the start and end columns and start and end lines that the span covers.[5]

There are located versions of many of the AST types. For example, `HsExpr` is the type that represents expressions, and there is a related type `LHsExpr` that represents a located expression whose definition is shown in Figure 22.

The `HsExpr` type represents much of the Haskell language such as function application, lambdas, if and case statements. Pattern matches are represented by a `MatchGroup` type. Each `MatchGroup` contains a list of `Match`es and some typing information. Consider the case statement in Figure 23. This would be parsed into the `HsCase` constructor of `HsExpr`. This constructor is seen in Figure 24. The first constructor argument represents the expression that the patterns are matched against (the tuple `(x,y)` in this case). The abstract syntax tree of this expression is shown in Figure 25. The second argument to `HsCase` represents all of the matches in the case statement. `MatchGroup`s are used to represent any code that associates some patterns with a right hand side expression. Function bindings, lambda expression, and case statements all use `MatchGroup`s.

[5] GHC has small optimisation where if a span exists entirely on a single line it only stores the single line number and the start and end column, instead of storing the same line number twice.

```
case (x, y) of
  (Just i, Just j) -> Just (i+j)
  (Just _, Nothing) -> x
  (Nothing, Just _) -> y
  (Nothing, Nothing) -> Nothing
```

Figure 23: A case statement

```
HsCase (LHsExpr id) (MatchGroup id (LHsExpr id))
```

Figure 24: The `HsCase` constructor

In the case expression from Figure 23 each of the four pattern matches is a `Match` in the expressions `MatchGroup`. Each `Match` associates a left hand side pattern with a right hand side clause.

3.2.3.2 The syntax tree

The previous section gave a brief overview of GHC's identifiers and its representation of expressions. This section describes the broader picture, the representation of whole Haskell programs.

According to GHC a Haskell program is simply a list of modules.[6] Inside of GHC each module is represented by an `HsModule`. This top level structure keeps track of everything that the module imports and exports, and a list of all the declarations that the module defines. Declarations (of type `HsDecl`) are what represent everything that can be defined in Haskell.

The `HsDecl` type is used as a wrapper around other types. Bindings (of functions and/or values), instance, and type class declaration are differentiated by `HsDecl`'s constructors. A few of the more common `HsDecl` constructors are shown in Figure 26.

`HsDecl` provides a high level categorisation of what type it is. There is very little information stored at this level, the inner type is the "payload" of the declaration (e.g.

[6]This is a very simple view and additional tools are needed to properly represent "projects" which is the context that most Haskell programs exist inside of. This is discussed further in Section 3.5.1.

```
(L {case.hs:(3,11)-(7,31)}
             (HsCase
              (L {case.hs:3:16-21}
               (ExplicitTuple
                [
                 (L {case.hs:3:17}
                  (Present
                   (L {case.hs:3:17}
                    (HsVar
                     (Unqual {OccName: x})))))),
                 (L {case.hs:3:20}
                  (Present
                   (L {case.hs:3:20}
                    (HsVar
                     (Unqual {OccName: y})))))])))
```

Figure 25: The fragment of parsed abstract syntax representing the expression being pattern matched in Figure 23.

```
data HsDecl id =
    TyClD (TyClDecl id)
  | InstD (InstDecl id)
  | ValD (HsBind id)
```

Figure 26: A subset of `HsDecl constructors`

`HsBind` or `TyClDecl`). These payload types are what store the type and expression-level abstract syntax of Haskell that was described in the previous section.

Some notable payload types are `TyClDecl`, `InstDecl`, `HsBind`, and `Sig`. `TyClDecl` represents a family, type synonym, data, or type class declaration. The `InstDecl` is used to represent the instance declarations for type classes, and data or type families. The real workhorse of the language is `HsBind`. `HsBind` represents functions and pattern bindings (e.g. `(x,y) = (1,2)`). The `HsBind` type also has constructors that are introduced by the type checker such as dictionary binding. Finally `Sig` is the data type that represents type signatures.

GHC's abstract syntax, for the most part, resembles the syntax described in the

Haskell 2010 report. Types in GHC's syntax may have additional constructors to support GHC's language extensions but the broader AST structure stays the same for most programs. The syntax for language extensions can be quite extensive and can make the documentation of the GHC API difficult to understand.

3.3 Generic programming

The need for a generic programming library, as previously discussed in Section 3.1.1, remains the same when using the GHC API's AST as opposed to Programatica's. Currently HaRe still uses Strafunski-StrategyLib as well as another library, Scrap Your Boilerplate. Scrap Your Boilerplate (SYB) is a generic programming library developed by Ralf Lämmel and Simon Peyton-Jones (Lämmel and Jones 2003).

SYB is included with GHC and has become the Haskell community's standard generic traversal library, however HaRe still uses Strafunski because it provides "stop" traversals. These traversals (stop-top-down, type preserving, and type unifying) descend the tree until the strategy "succeeds" (that is, the type of the transforming function's parameter is found) and then stops. Essentially it cuts a line across the abstract syntax tree where everything below the line is unvisited. This traversal is useful when refactoring because of how the ASTs types are nested within each other. For example, when a refactoring modifies the body of a function the stop traversals ensure that when the transformation is applied, it is applied with the expression representing the whole binding rather than sub-expressions.

3.3.1 Generic Traversals

The Stratego/XT library was one of first systems for programming tree transformations in a systematic way (Visser 2004). Stratego developed the idea of a transformation strategy. A strategy is the combination of a term rewriting function and a traversal function that describes how that rewriting function should be applied to a tree of terms.

```
type Name = String

data Expr =
    Value Int
  | Var Name
  | Add Expr Expr
  | Assign Name Expr
    deriving(Data,Typeable)
```

Figure 27: A simple expression type.

Stratego provides combinators that help construct term rewriting functions and tree traversal functions (Visser 2004).

Stratego is an untyped transformation system and so was difficult to transform statically typed Haskell in a way that respects its types. Strafunski was "largely inspired" by Stratego, but for a statically typed context (Lämmel and Visser 2002).

3.3.2 Scrap Your Boilerplate

Scrap Your Boilerplate (SYB) is a generic traversal library for Haskell that was inspired by Strafunski (Lämmel and Jones 2003). SYB comes with GHC and has become the most used generic programming library in the Haskell community.[7]

Suppose there was a simple expression language that contained integers, integer addition, assignment, and variables. This language is represented by the type defined in Figure 27 and a function that works over this type to rename a variable "x" to "a" is defined in Figure 28.

There are four cases to the `renameXVar` function. The first case matches when `renameXVar` is called with a `(Var "x")` value. This is one of the cases that the function will replace the "x" name with an "a" name. The second case matches an assignment and is the other case when actual "work" happens. In this case the function needs to check if the variable being assigned is an "x" or not, if it is then the name being

[7] According to `http://packdeps.haskellers.com/reverse`

```
1 renameXVar :: Expr -> Expr
2 renameXVar (Var "x") = Var "a"
3 renameXVar (Assign c e) =
4   | c == "x" = Assign "a" (renameXVar e)
5   | otherwise = Assign c (renameXVar e)
6 renameXVar (Add e1 e2) = Add (renameXVar e1) (renameXVar e2)
7 renameXVar v = v
```

Figure 28: A function to rename the "x" variable.

assigned is replaced with `"a"` otherwise the name remains the same. Then there is a recursive call replace the name in the right hand side of the assignment. The other two cases will not directly modify an expression. In the `Add` case there is a recursive call to perform replacements in its sub-expressions. The final case is a catch-all term that returns its parameter. This will be called when `renameXVar` is called with a `Value` or a `Var` that is not `"x"`.

This is fairly straightforward and doesn't take much time to write. However, if subtraction was added to the definition of expression `renameXVar` would need to be updated as well to include a recursive call very similar to the addition case. These duplicated recursive calls are just boilerplate code (Lämmel and Jones 2003). In this small example having a few of these types of cases is not an issue. However, as the expression type begins to approach the size of an actual programming language writing traversals like `renameXVar` would become much more time-consuming and a nightmare to maintain.

The reduction of boilerplate code like this is the point of SYB. SYB allows us to rewrite `renameXVar` as shown in Figure 29.

This example nicely illustrates the four key components of an SYB traversal (Lämmel and Jones 2003).

- The function that performs the "interesting" part of the traversal
- A type extension for that function

```
1 import Data.Generics
2
3 rename :: Name -> Name
4 rename "x" = "a"
5 rename n = n
6
7 renameXVar :: Expr -> Expr
8 renameXVar = everywhere (mkT rename)
```

Figure 29: `renameXVar` written using SYB

- A generic traversal combinator
- The data type to be traversed must be an instance of the `Typeable` and `Data` classes (as explained below)

From the earlier example the "interesting" part of this traversal is the rename function, because this function contains the code that actually changes the name "x" to the name "a." The `mkT` function extends the type of the `rename` function to `Typeable a => a -> a`.

Type extension allows for the `rename` function to work over any members of the `Typeable` class rather than just `Name`s. The extended version of `rename` will work as expected when provided with an argument of type `Name`, and if an argument of any other type is provided the traversal will continue to descend the tree by moving onto the argument's children, if it has any, and returns the argument unchanged if it doesn't.

The `everywhere` function is this traversal's generic combinator. `everywhere` will apply a generic function to every node in a tree in a bottom-up manner, as long as that tree is of type `Data` and `Typeable`. A member of the `Typeable` class has defined a generic representation of itself and members of the `Data` class implement generic folding operations. As you can see that the `Expr` data type, from Figure 27, derives both the `Typeable` and `Data` classes so it can be traversed by `everywhere`.

```
1 bVars :: Expr -> [Name]
2 bVars e = everything (++) ([] `mkQ` f) e
3   where f (Assign nm _) = [nm]
4         f _             = []
```

Figure 30: A generic function that collects all bound variables from an expression.

3.3.2.1 Types of Generic Algorithms

SYB defines three types of generic algorithms: transformations, queries, and monadic transformations. The `rename` example from the previous section is an example of a transformation. Transformations preserve the type of the structure that is traversed. Queries, on the other hand, are "type unifying" algorithms. Queries are good for summarizing information contained in a data structure. A query would be used, for example, to traverse an expression and collect all of its bound variables. Using the same expression type (Figure 27) from the previous section, the following function extracts all bound variables from a given expression.

`everything`, as seen in the `bVars` function from the previous listing, is the generic query combinator that queries all nodes, top down from left to right (SYB Package 2014). The first argument to `everything` is the function it uses to combine separate results from the query. In this case, the lists of names will be appended together. The “interesting” function `f` that actually returns a singleton list of type `Name` when applied to a variable binding, and an empty list otherwise. `f` has been extended with the `mkQ` function; `mkQ` will apply `f` when possible, otherwise it will just return a default value, in this case the empty list.

It is also useful to perform monadic transformations. A simple example of this would be to rewrite the finding bound variables example from previously but instead the list of found results is stored as a piece of state.

An advantage of monadic traversals is that both querying and transformations can happen in a single pass. The example in Figure 32 renames every `Name` in an expression by adding `"_old"` as a suffix and stores the original names in a list.

```
type TransformState = State [Name]

bVars :: Expr -> [Name]
bVars e = execState findVars []
  where findVars = everywhereM (mkM f) e
        f :: Expr -> TransformState Expr
        f e@(Assign n _ ) = do
          modify (\lst -> n:lst)
          return e
        f e = return e
```

Figure 31: Finding bound variables using the state monad

```
renameVars :: Expr -> TransformState Expr
renameVars e = everywhereM (mkM f) e
  where f :: Expr -> TransformState Expr
        f (Assign n e) = do
          modify (\lst -> n:lst)
          return (Assign (n++"_old") e)
        f (Var n) = return (Var (n++"_old"))
        f e = return e
```

Figure 32: Changing every found name and storing the old names in a list.

This type of generic traversal is very common in HaRe because traversals often need to make use of a refactoring's stored state or run something from the GHC API which needs the features provided by an instance of `GhcMonad`, all while modifying the abstract syntax. The GHC API's operations all work within the `GhcMonad` that provides the features GHC needs to compile a single Haskell source file such as IO, logging warning, exception handling, and keeping track of the compilation session (GHC API 2016).

3.4 ghc-exactprint

After transforming the abstract syntax tree with generic programming, that abstract syntax needs to be printed. A challenging part of building a refactoring tool is that a user

```
1 f a = (a+ 1 )
```

Figure 33: A definition with strange spacing.

will not want non-refactored parts of their code to change at all. A source program's layout and comments need to be preserved after refactoring.

Prior to GHC version 7.10.1, the location of certain keywords and punctuation (such as `do` and `let`) and user comments were lost after parsing. This made parsing and then printing an exact copy of a GHC Haskell source file impossible from the AST alone. GHC's 7.10.1 release added annotations for the "lost" syntax elements that had previously not been represented in the parsed abstract syntax (Zimmerman 2015).

The parsed source now includes a map that associates each "lost" keyword with the source span that the keyword can be found in, and that keyword's exact location. This approach was taken to avoid littering the existing AST with functionally meaningless keyword data (Zimmerman 2015).

Even with the position of every syntax element being recorded, printing a module after the AST has been modified is not easy. Many AST elements are "located" with a source span that indicates that element's exact position in the file. This means that any change to the AST will require updating the location of all the syntax elements that occur later in the line at least for single line changes and, in the case of changes that modify entire lines, every element after that change will need its location to be updated.

Ghc-exactprint simplifies this immensely by allowing us to position elements relative to their neighbours rather than absolutely (Zimmerman and Pickering 2016). After parsing a source file, HaRe takes the annotations GHC returns with the parsed abstract syntax and relativises them using ghc-exactprint. For each syntax element ghc-exactprint creates a new data type called an "`Annotation`" which contains an offset that indicates where this element should be positioned compared to the previous element.

Take for example the definition in Figure 33. The absolute position of the plus sign

is row one column nine (GHC's locations are one-based) but using ghc-exactprint we instead can think of the position of the plus sign using the offset `(0,0)` because there is no space between it and the previous element (the variable `a`). Using this system the number literal following the plus sign has an offset of (0,1) because of the single column of space before it.

The `Annotations` are stored in a map that is keyed on the parsed location of a syntax element and the string representation of the AST constructor. In the previous example the right hand side of the definition is located at the source span (1,7)-(1,13) which stands for "row one columns, seven through thirteen" and GHC represents this expression with the `HsPar` constructor for the `HsExpr` type. The source span and the constructor together can be combined to retrieve the annotation data associated with this fragment of the AST. This syntax tree has two elements associated with it, the opening and closing parenthesis. Each of these keywords is given its own offset, which in this case is (0,0) for the opening parenthesis[8] and (0,1) for the closing parentheses. We could infer from the use of the `HsPar` constructor that this tree is wrapped in parentheses however the user's specific spacing would be lost without the annotations.

Comments are another element of a source file that prior to GHC 7.10.1 were "lost" after parsing. Using ghc-exactprint comments are stored in the annotations associated with the next piece of syntax. In the code snippet in Figure 34 the comment on line three is added to the annotations associated with the declaration of the function `f` along with a delta position that indicates the comment is a single line before this declaration starts.

3.5 The current implementation of HaRe

We have just discussed the major components on which HaRe depends. The ghc-api gives access to the internal representation of Haskell syntax, the generic programming

[8]The offset in this case is (0,0) and not (0,1) as you might expect because the space between the equals sign and the opening parenthesis is represented in the offset for the entire right hand side expression.

```
1 type Name = String
2
3 --a comment
4 f i = i + 1
```

Figure 34: The comment is "attached" to the AST of the function `f`.

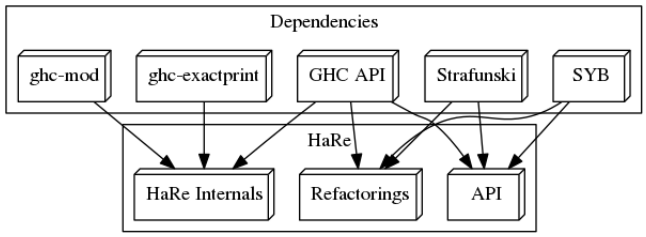


Figure 35: A diagram of the components of HaRe and their dependencies.

libraries Scrap Your Boilerplate and Strafunski-StrategyLib allow HaRe to work more easily with that internal representation, and ghc-exactprint is how HaRe preserves the source files spacing when writing the output file. The relationship between the dependencies and HaRe's components is shown in Figure 35. From this foundation the focus of HaRe can be on refactoring rather than solving these more generic problems.

Very little, if any, of the code from the original implementation of HaRe is used today. Development of HaRe has been coordinated through GitHub[9] since 2011. When I began working on HaRe in 2013, Alan Zimmerman had upgraded much of the internal structure of HaRe to use the GHC API.

The internal structure of HaRe is mostly the work of Alan, however a description of the current implementation has not appeared in the literature yet and so it has been described in some detail in this section. My contributions to HaRe are in the development of the refactorings described throughout this thesis and in the creation of a high level API for program analysis, synthesis, and transformation which is described in Section 5.3.

[9] `https://github.com/RefactoringTools/HaRe`

```
1 newtype RefactGhc a = RefactGhc
2    { unRefactGhc :: GM.GhcModT (StateT RefactState IO) a}
```

Figure 36: HaRe's Monad `RefactGhc`

3.5.1 The Internal Structure of HaRe

The GHC API exposes the functionality needed to compile a set of Haskell source files. In reality Haskell programs consist of more than just single source files without dependencies. Projects are organised using build tools such as Cabal (Jones and Coutts 2016) or Stack (Stack contributors 2017) that handle acquiring dependencies with the correct versions, running tests, compilation, and packaging binary code into an executable for the programmer. HaRe needs to be aware of the context that these tools provide because refactorings may change multiple modules or modify modules that import modules from external dependencies. In HaRe's case, ghc-mod provides a monadic context that handles these build environments and compiler setup (Gröber 2017). Within ghc-mod's context, HaRe keeps track of the state of the refactoring session.

Figure 36 shows the definition of HaRe's monad that each refactoring runs in. `RefactState` is an algebraic data type that keeps track of all the settings for the refactoring session, the abstract syntax trees of the current target file, and filepath for refactoring a single file.

Even though the GHC API produces a separate version of AST for each compilation stage (parsing, renaming, and type checking), the annotations are only included in the parsed AST. Once a refactoring has finished, all of its changes need to be reflected in the parsed abstract syntax tree and its annotations because this is the structure that ghc-exactprint works over. HaRe's state still contains all three of the syntax trees because the renamed and typed sources are useful for the additional type and scope information they contain about a source file.

3.5.2 HaRe's API

Modifying and reasoning about GHC's abstract syntax and maintaining the associated annotations is still a complex task, even with help from HaRe's dependencies. HaRe defines its own API to help fill this gap between its dependencies and its refactorings. In addition to the obvious functions that are required for running a refactoring within the `RefactGhc` monad, the API also includes helper functions that make working with the state easier.

HaRe also defines a large collection of program analysis and transformation functions. For example, pulling the binding of a top level variable from a module's abstract syntax tree is a task that many refactorings have to do, so this functionality is part of HaRe's API. There are also small program transformations that are not in and of themselves refactorings but common low-level modifications that are useful to several refactorings, such as adding a new import declaration or making a function infix by wrapping it in back quotes (the ` character).

Additionally there are several transformations that do not affect the abstract syntax, but which change the annotations that format ghc-exactprint's output. Adding new lines before a syntax element doesn't change the meaning of a program but is important for a refactoring's output to be well formatted and easy to read.

HaRe's dependencies help abstract away the low-level[10] details of a language backend, build tools, and pretty printing. HaRe's API tries to close the gap still left between the dependencies and the refactorings themselves. In the next section we will take a look at how HaRe's refactorings are implemented.

3.5.3 Implementing Refactorings in HaRe

HaRe does not enforce any particular structure onto a refactoring implementation. A refactoring implementation in HaRe needs to do a few things:

[10] Low level from the perspective of a refactoring at least.

1. A refactoring must run inside the `RefactGhc` monad whose state is where the abstract syntax of a target module is stored.

2. A refactoring is also expected to return a list of "`ApplyRefacResults`" which contains an updated parsed AST and annotations, together with the filepath the AST originated from; this updated AST is what HaRe writes out as the result of the refactoring.

Beyond those two features, refactorings are free to be implemented however the programmer chooses. However, certain conventions have been adopted within many of HaRe's refactorings. When describing a refactoring one would imagine that checking *pre*conditions would be the first thing the implementation of that refactoring computes. A more efficient implementation checks preconditions throughout computation alongside the AST transformation. For example, the renaming refactoring doesn't check for name clashes in client modules until after the source module has been modified. If a name clash is found in a client module then an error is thrown at that point and the whole refactoring is rolled back. Though this slows down the time to failure somewhat, it saves HaRe from traversing every client module twice, once for preconditions and a second time for rewriting.

Merging precondition checking with transformation saves the refactoring from traversing parts of the AST multiple times. For example, the renaming refactoring checks for name conflicts while it descends the AST, replacing the old name with the new one. Obviously this strategy only works for certain preconditions: the only precondition for deleting a definition is that the target definition is not used. The transformation only affects the syntax tree of the definition to be deleted so the implementation of the refactoring has to do a separate scan of the rest of the target module and any of its client modules to determine if the target definition is used or not.

3.5.4 Using GHC's Abstract Syntax Trees

Each stage of the compiler takes the previous stage's output and creates its own version of the tree decorated with some additional information. This provides a challenge when writing refactorings because all of the changes made to the source needs to be reflected in the parsed syntax tree. In general the parsed syntax contains enough information to perform a refactoring. Since the parsed AST is also where changes must be made for ghc-exactprint to output the modifications, most of the refactorings only look at the parsed AST. If information is required from one of the other compilation stages it is either pre-collected at the start of the refactoring, or the `SrcSpan`s that decorate all of the syntax trees can be used to navigate to the equivalent points in different syntax trees.

In the same way that a refactoring can be sped up through careful consideration of how much re-traversing of the abstract syntax tree is absolutely necessary, it is also important to be mindful of when it is absolutely necessary to traverse through the other stages' trees. For the most part HaRe's refactorings can avoid having to look up information in either the renamed or typechecked trees. The renaming refactoring, for example, uses a `NameMap` which associates every `RdrName` in the parsed source with its `Name` from the renamed source. This allows the refactoring to ensure that the instance of a variable it is going to rename is actually the targeted variable and not just a variable with the same name in a different scope. This map is calculated at the start of the refactoring, so the renamed syntax tree only needs to be traversed once rather than multiple times to check if each instance of a name that matches the target variable's name actually refers to the target variable or if the same name is used again in a different scope.

Collecting any required information from the other syntax trees up front, is a good way to limit the amount that these trees are traversed, but often traversing these trees can be entirely avoided. For example, the "generalising `Maybe`" refactoring described in Chapter 4 works on replacing the `Maybe` type with a more generic type if possible.

`Maybe a` is a type that represents a computation that may return a value of type `a` or fail in some way, this case is represented by a `Nothing` constructor. One tricky part about Haskell is that imports can be "qualified," that is certain modules can force the values from another module be referred to with a prefix, then variables with the same name as an imported value can be used without name clashes. This means that it is not safe to just assume that any constructor named `Just` or `Nothing` is actually of type `Maybe`. The generalising `Maybe` refactoring, does not check the type of all the `Just`s and `Nothing`s encountered in the target program, instead the refactoring begins by checking the imports of the target module to see if the `Data.Maybe` module has been given a qualification and takes into account that the target module will refer to `Maybe`'s constructors with a qualified name. After that check, since `Maybe` is included in the prelude it is safe to assume that any `Just`s and `Nothing`s in the target program are of type `Maybe`.

One refactoring that does need to reference the typed syntax tree fairly heavily is the List to Hughes List refactoring described in Section 4.5. This refactoring needs to change the type of entire subtrees, so it references the typed syntax tree to see what the type of the current node is during the transformation. This is fairly straightforward due to the source being tagged with `SrcSpan`s that are consistent throughout the compilation process. This means that a piece of typed abstract syntax is the same as a particular parsed syntax subtree if their `SrcSpan`s are the same. Searching through the later compilation stage's trees is as simple as doing a full traversal of the tree searching for the appropriate `SrcSpan`.

3.6 Summary

The work done for this thesis was fortunate enough to be built upon the contributions of others who made HaRe and its dependencies. The focus of this chapter has been the

current implementation of HaRe[11] and some of its notable dependencies.

The next chapter will begin to describe the major contribution of this thesis: the design and implementation of data-driven refactorings. The next chapter will begin the discussion of what is a data-driven refactoring and some simple refactoring designs.

[11] Which for the purpose of this thesis is release v0.8.4.1 on GitHub: `https://github.com/RefactoringTools/HaRe/tree/4055ef45f0de3c966fd7841986ab0ed2ee814055`

Chapter 4

Data-driven refactorings

Some refactorings focus on rewriting the structure of a program (e.g. refactoring an `if` to `case` statement), but a data-driven refactoring focuses on the data that a program works over and *how* that program manipulates that data. Any changes that a data-driven refactoring makes to the structure of the program are *driven* by the data types of that program rather than being the main motivation of performing the refactoring.

Consider the introducing a type synonym refactoring (that is described in more detail in Section 4.3), this refactoring adds a type synonym, i.e. a user chosen name, for some type. The motivation behind this is to ensure that the names of types correspond to the types that they represent. For example, if some strings are used to represent customer names in a program, introducing a synonym for strings called `Name` helps clarify what kind of data certain strings are supposed to represent. In this way the refactoring's changes to the source program are motivated by the data in that program and the desire for the program's types to better communicate the nature of that data.

This chapter covers, more specifically, what is meant by data-driven refactorings by example. It begins with a discussion of data-driven refactorings for imperative and object-oriented languages in Section 4.1. Imperative programs are written as a sequence of steps that modify values to a desired output state. Types in an object-oriented program exist in a strict hierarchy, every type is a sub-type of a top type commonly known

as "`Object`." These two factors mean that object-oriented data-driven refactorings modify the hierarchy, by moving methods between related classes or extracting new child or parent classes, while leaving the structure of the code mostly unchanged. The rest of this chapter will describe data-driven refactorings for functional programs.

In comparison to imperative programs, functional programs describe the relationship between the input and output data. Functional programming languages also tend to offer a much richer type system than object-oriented languages where data types are typically either classes or a non-user expandable set of primitive types. These two facts mean that the structure of a functional program can be determined by the types it works over to a much greater degree than in imperative languages.

Section 4.3 describes the "introducing a type synonym" refactoring. This is the renaming of data-driven refactorings: neither the structure nor the type of the target program changes, just how the program refers to the types it works over. Next, Section 4.4 describes a generalisation refactoring. This thesis outlines two generalisations, one (described in Section 4.4) makes code of a specific type (`Maybe`) work over a type class (`MonadPlus` or `Monad`) instead. The other generalises code using the functions defined in a particular type class to instead use the functions of it's superclass (covered in Chapter 6). Finally Section 4.5 describes a refactoring that tranforms programs that work over lists to instead use an alternate implementation of lists. The refactoring method described in this chapter could be applied to refactor between any types that can be "projected" onto another type. A type "`A`" can be projected onto some type "`B`" when all of the information stored in an instance of `A` can be transformed to be of type `B` instead.

4.1 Object-Oriented Data Refactorings

The origins of refactoring are deeply rooted in the object-oriented world (Griswold (1992); Opdyke (1992)), though its origins can be found in work on transforming

Algol (Burstall and Darlington 1977). The canonical catalogue of refactorings remains Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Fowler 1999). Fowler's catalogue of refactorings are all written in Java though he purposefully avoided using any features that were unique to Java, so that the refactorings could be applied in many different object-oriented programming languages.

As a functional programmer going through this catalogue there seem to be three types of refactorings.

- Refactorings that are directly applicable in a functional language
- Refactorings that could be adapted for use in a functional language
- Refactorings not applicable to a functional language

That first type of refactorings are typically structural refactorings and their usefulness to a functional program is easily understood. Refactorings like renaming or adding a parameter don't depend on object-oriented features existing in the target language.

The third type of refactorings are so dependent on features associated with object-oriented languages that they are impossible to implement or have no equivalent in a functional language. The "remove setting method" refactoring depends on the common OO pattern of each field of a class having "getter" and "setter" methods that retrieve or modify that field respectively. This pattern, on the other hand, is not as common in functional languages because most do not support objects, and immutability makes a "setter" function irrelevant. OCaml would be a notable exception to this rule: OCaml supports objects and allows the programmer to mark variables as mutable, and so getter and setter methods are possible.

The OCaml object from Figure 37 would be a valid target for this refactoring. However these methods are not as common as they are in imperative object-oriented languages with mutable data as the default, so a refactoring to remove a setter method is of limited applicability even for OCaml.

```
let mInt init_i = object
   val mutable i = init_i

   method get_i = i
   method set_i new_i =
     i <- new_i
 end
```

Figure 37: An OCaml object with getter and setter methods.

The refactorings that could be adapted from Fowler (1999) are much more interesting to a functional programmer. The specifics of these refactorings are not directly applicable to functional programs but the underlying motivations are relevant to any programming paradigm.

For example, the motivation for the "Replace Data Value with Object" refactoring in (Fowler 1999, pg. 175) reads as:

> Often in early stages of development you make decisions about representing simple facts as simple data items. As development proceeds you realize that those simple items aren't so simple anymore. A telephone number may be represented as a string for a while, but later you realize that the telephone needs special behavior for formatting, extracting the area code, and the like. For one or two items you may put methods in the owning object, but quickly the code smells of duplication and feature envy[1]. When the smell begins, turn the data value into an object.

This refactoring extracts a field that was some primitive type into an object. The example from (Fowler 1999) works over an order class with a string that represents the customer that placed an order.

The refactoring creates a new customer class that has a string field with a getter method as seen in Figure 39. The customer class doesn't add any additional features

[1]Feature envy is when the methods of one class are more concerned with the features of another object than the one it is in. In this phone number as strings example feature envy would be seen when objects are doing the area code and formatting calculations for the object that contains the phone number.

```
1  class Order {
2     public Order (String customer) {
3        _customer = customer;
4     }
5
6     public String getCustomer() {
7        return _customer;
8     }
9
10    public void setCustomer(String arg){
11       _customer = arg;
12    }
13
14    private String _customer;
15 }
```

Figure 38: The Order class

but the extra layer of abstraction that has been added sets up the code base for further development. The customer object could have fields added that represent contact info or further demographic information without polluting the order class with order-irrelevant data.

Functional programmers have to make similar data representation decisions to those the object-oriented programmer make. At the start of a project representing a customer by their name could be reasonable. As the project develops this can become a serious limitation and a more robust abstraction could be required.

This section introduced how object-oriented refactorings help build up the data model over the lifetime of a project. These sorts of refactorings for a language with a rich type system like Haskell offer many more choices for evolving the data model of a system. Though a Haskell programmer can take inspiration or directly reimplement several of the canonical Fowler refactorings there may be additional refactorings that are completely functional with no OO counterparts, like monadification (Chapter 7) or abstraction of expressions into a local `where` or `let` clause. The rest of this chapter will describe refactorings for Haskell that support data model evolution.

```
1  class Order {
2     public Order (String customer) {
3        _customer = new Customer(customer);
4     }
5
6     public String getCustomer() {
7        return _customer.getName();
8     }
9
10    public void setCustomer(String arg){
11       _customer = new Customer(arg);
12    }
13
14    private Customer _customer;
15 }
16
17 class Customer {
18    public Customer(String name){
19       _name = name;
20    }
21
22    public String getName() {
23       return _name;
24    }
25
26    private final String _name;
27 }
```

Figure 39: The result of the Replace Data Value with Object refactoring when applied to the customer field of the order class.

4.2 Data-Driven Refactorings in Haskell

Haskell offers a rich environment for data representation. The Haskell 2010 standard defines several types that are included in the prelude, including tuples, lists, characters, strings (which are just lists of characters), several types of numbers, and the function type. GHC also provides more types than the standard requires and programmers can construct new algebraic and abstract data types or rename existing types with type synonyms. Type classes support overloading as well. The standard library of GHC

comes with many type classes that can help produce powerful abstractions (Yorgey 2009b). Contrast this with how most object-oriented type systems are either unified, where every type is a subclass of some top level `Object` class, (e.g. C# or Ruby) or there are a fixed set of predefined primitive types as well as the object hierarchy (e.g. C++ and Java's type systems).

These two different approaches to type systems both have pros and cons, which have sparked a vigorous (and quite protracted) debate. The goal of this thesis is not to add to this debate, instead it is to build from a number of general principles that transcend the debate.

- Data representation is a language independent problem that must be answered in every project.
- The way a project manages and represents its data is very likely to evolve over a project's lifetime and this is indeed desirable.
- Refactoring is a structured way to support this evolution.
- Refactorings for a language like Haskell need to take a different approach than those for object-oriented languages, particularly when refactorings are data-driven rather than "structural."
- Data representation in a rich type system like Haskell's, in some sense, determines the structure of the program.

4.3 Introducing a Type Synonym

Type synonyms in Haskell, as mentioned previously, are a way to name an existing type. A simple example can be seen in Figure 40.

Any place that where the `Foo` synonym is in scope the new name can be used to refer to any value of type `(String, Int)`. In fact "`Foos`" and "`(String, Int)`s"

```
type Foo = (String, Int)

f :: Foo -> Foo
f x@(_, 0) = x
f (str, i) = (tail str, i-1)
```

Figure 40: A simple type synonym.

```
data Order = Order {customer :: String}

numberOfOrdersFor :: [Order] -> String -> Int
numberOfOrdersFor orders name = length (filter (\ord -> name
    == (customer ord)) orders)
```

Figure 41: Using an algebraic data type for `order`.

are completely interchangeable. Introducing a synonym is a good way to quickly and simply name types to suggest their specific use in the current application.

Returning to the example, used in Section 4.1, of an order type that keeps track of the customer who placed the order used in, from Fowler (1999), Figure 41 shows a Haskell implementation of this type and a function that counts how many orders a particular customer has placed in a list of orders.

The initial representation of a customer as just a `String` is underdeveloped. This representation of a customer only allows the system to store a single "field" of information per customer, as the system becomes more developed it will want to associate more information with a particular customer and this representation will need to change.

Introducing a customer synonym is a simple step that sets up the code base for further development. The synonym will clearly mark which strings in the program stand for customers and which do not.

The "introducing a type synonym" refactoring works by taking a type and a valid synonym name (as per the Haskell 2010 standard (Marlow et al. 2010)) and creates a new synonym. In this case the type is `String` and the synonym name should be something like "`Customer`." The (only) precondition of the refactoring is that the

```
1 type Customer = String
```

Figure 42: The customer synonym

```
1 printThankYou :: String -> Order -> IO ()
2 printThankYou businessName order = do
3   putStrLn ("Thank you " ++ (customer order) ++ " for your
       order.")
4   putStrLn (businessName ++ " hopes to see you again soon!")
```

Figure 43: The `printThankYou` function

new synonym's name cannot cause a name clash with any other types. Also if the synonym is exported to any client modules that define something with the same name, these modules will need to qualify the import of the synonym.

After this, the next part of the refactoring involves replacing the appropriate uses of the type with the new synonym. This part of the refactoring is difficult to automate and needs to be interactive, which is not a feature of HaRe yet. The current implementation of this refactoring takes in a flag as an argument that determines if the refactoring will replace all instances of the type with the newly introduced synonym. This is not the ideal implementation of the refactoring, but there is no way to infer which instances of `String` (in the case of this example) should be replaced with the `Customer` synonym. The code from listing 41 can have all of the string instances replaced by `Customer` because all the strings are being used to represent a customer. However, consider the `printThankYou` function in Figure 43 which has type `String -> Order -> IO ()` and prints out a customized "thank you" message to the customer for their order.

Though the first argument to `printThankYou` is a `String` it does not represent a customer therefore should not be replaced by the new synonym. The implicit meaning of the first argument hasn't been encoded in the type system so the programmer has to make a decision during the refactoring to make their intention for each instance of `String` clear. The final result of the refactoring can be seen in Figure 44.

```
1 type Customer = String
2
3 data Order = Order {customer :: Customer}
4
5 numberOfOrdersFor :: [Order] -> Customer -> Int
6 numberOfOrdersFor orders name = length (filter (\ord -> name
    == (customer ord)) orders)
7
8 printThankYou :: String -> Order -> IO ()
9 printThankYou businessName order = do
10  putStrLn ("Thank you " ++ (customer order) ++ " for your
      order.")
11  putStrLn (businessName ++ " hopes to see you again soon!")
```

Figure 44: The program after adding a synonym for String.

This transformation might seem like too small of a step. Wouldn't it be preferable to introduce a more powerful abstraction such as an algebraic data type? One of the principles of the Agile Manifesto is "simplicity" which is described as "maximizing the amount of work not done is essential" (Fowler and Highsmith 2001). The work not done in this case is the introduction of a more complex customer representation. This small step does clearly differentiate the strings that represent customers from other strings that represent other types of data.

Small refactorings are also a good practice for tool builders. Creating multiple small composable steps that can be used in multiple ways is much more flexible than large "monolithic" style refactorings.

4.4 Generalisation

A common type of refactoring is generalisation. A generalising refactoring that is structural would be the "generalise a definition" refactoring from (Li 2006). This refactoring takes an expression from inside of a function and refactors it so that expression is passed as a parameter instead. Figure 45 shows two functions, `f` and `f_ref`, where `f_ref` is the function `f` after the generalise a definition refactoring is performed on it. The

```
1 f :: Num a => a -> a -> a
2 f x y = x + y
3
4 f_ref :: Num a => (a -> a -> a) -> a -> a -> a
5 f_ref op x y = x `op` y
```

Figure 45: Generalising the + operator from the function f.

refactored function has become more general because instead of only being used for summing two numbers, it can be reused for any binary operations on numbers.

In an object oriented language generalisation refactorings deal mostly with either "moving methods around a hierarchy of inheritance," or creating new classes that change the hierarchy (Fowler 1999). Examples of generalising refactorings for OO languages as described in (Fowler 1999) are pull up method, push down method, extract subclass, and extract interface. Interestingly about half of the generalisation refactorings described by Fowler are better described as specialisations, such as push down method, because they make general code more specific by moving it further down the object hierarchy.

In object oriented languages, moving elements "up" the hierarchy (as in from subclass to superclass) is a generalisation, and doing the opposite is specialisation. This should make intuitive sense; there are more fruits than just oranges, so moving a method from the orange class up to its superclass, the fruit class, makes that code more general because it can now be applied to all subclasses of fruit, not just the orange class.

Code in functional languages doesn't inhabit the rigid hierarchy of objects that exists in OO languages, so generalisation can take on more forms. The core effect that a generalisation refactoring has is taking something specific (such as a sub-expression, or an instance of a particular type) and makes it more general in some way. A sub-expression may be extracted into a parameter which makes the target function more general because the behaviour that was previously set by the extracted sub-expression

is now determined by the expression passed in via the new parameter. Specific instances of types are generalised by being replaced with a type class. The purpose of type classes is to provide a generic interface for some behaviour, so replacing a specific type with a type class means that the target code is more general because now it can be applied to the set of types that implement that type class, not just a single type.

This section describes refactoring functions that use the concrete `Maybe` type to instead use one of the type classes it is an instance of, either `Monad` or `MonadPlus`. These refactorings will take a function of type `a -> Maybe b` and rewrite it to become either `Monad m => a -> m b` or `MonadPlus m => a -> m b`. These refactored types are a generalisation of the original function because instead of only being applicable to a single type (`Maybe` in this case) the refactored function can now be used with data of type `Monad` or `MonadPlus`.

4.4.1 Generalising Maybe

This section describes a refactoring that rewrites programs of type `Maybe a` to use the `MonadPlus` type class or, if possible, the `Monad` type class. `Maybe` is a data type whose declaration is shown in Figure 46. `Maybe` represents a computation that can fail or return a value of type `a`. When a computation has failed `Maybe` represents this "result" with the `Nothing` constructor. When a calculation succeeds then `Maybe` returns the result wrapped with the `Just` constructor. `Maybe` is a commonly used type in Haskell programs, it also implements many common type classes including `Monad` and `MonadPlus`.

The definition of `Maybe` as well as its instance declarations for the `Monad` and `MonadPlus` type classes are shown in Figures 46 and 47. The declaration of the `MonadPlus` type class as well as `Maybe`'s definition of it are shown in Figure 48.

```
data Maybe a = Nothing
             | Just a
```

Figure 46: The `Maybe` data type declaration.

```
instance Monad Maybe where
  return = Just

  (Just a) >>= f = f a
  Nothing >>= _ = Nothing
```

Figure 47: `Maybe`'s monad instance definition

4.4.2 Refactoring Maybe to MonadPlus

`MonadPlus` is the obvious generalisation of `Maybe`. `Maybe` encodes failure using the `Nothing` constructor and its definition of `mplus` chooses to discard failed computations on the left in exchange for a possibly successful one. This behaviour is exactly what `MonadPlus` is designed to encapsulate. This section describes a refactoring for rewriting functions that use `Maybe` to use the `MonadPlus` type class instead.

The function `showNat` takes a pure value and returns a `Maybe`, instead of taking in a `Maybe` and returning another `Maybe` as in the previous example. The more general version of `showNat` needs to be able to express the idea of failure that `MonadPlus` encodes with the `mzero` operation.

Figure 50 shows the refactored version of `showNat`. In this case the refactoring can simply replace the `Maybe`-specific calls with more general ones. This is very similar to the way that the expression that made up the anonymous function from Figure 54 was constructed by changing calls to `Just` to `return` and calls to `Nothing` to `mzero`.

Due to the `return` operator of `Monad` being equivalent to `Just` and `mzero` being supplied by `MonadPlus`, both of `Maybe`'s constructors can be replaced with more general operations. Between these two type classes all of the functionality of `Maybe` can be replaced in functions that build `Maybe`s. However if a function deconstructs a `Maybe` type through pattern matching it may not be generalisable. Consider the

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` r = r
  l       `mplus` _ = l
```

Figure 48: The `Maybe MonadPlus` instance declaration, and the `MonadPlus` class definition.

```
showNat :: Int -> Maybe String
showNat i =
 if (i >= 0)
   then (Just (show i))
   else Nothing
```

Figure 49: `showNat`

function `printResult` in Figure 51. `printResult` performs a pattern match over the `Maybe` type constructors rather than only in right hand side expressions; there is no way to generalise pattern matching on `Just`.

4.4.3 Refactoring Maybe to Monad

In some cases it is possible to generalise a `Maybe` type to become `Monad` instead. When functions duplicate the work that is done by `Maybe`'s implementation of bind then that function can be generalised to use the monadic interface instead.

The function `inc` from Figure 52 can be rewritten to use the monadic operations bind (»=) and `return` instead, this is because the definition of `inc` matches the definition of bind in the instance declaration of `Maybe` as a monad. Figure 53 has another version `inc` in which that relationship is clearer.

The new version of `inc` has been written using infix notation, to more closely match `Maybe`'s definition of bind. Also like `Maybe`'s bind when `inc`'s first argument

```
showNat :: (MonadPlus m) => Int -> m String
showNat i =
  if (i >= 0)
    then (return (show i))
    else mzero
```

Figure 50: `showNat` refactored

```
printResult :: (Show a) => Maybe a -> IO ()
printResult m =
  case m of
    Nothing -> putStrLn "Something went wrong"
    (Just i) -> putStrLn $ "The result is: " ++ (show i)
```

Figure 51: `printResult`

is `Nothing` it just returns `Nothing`. Finally the right hand side of `inc`'s second case was lifted into a separate function and is passed to the new `inc` as an argument, just like `bind`.

In practice the refactoring will, by default, create an anonymous function from the right hand side of the `Just` case with calls to `Just` and `Nothing` replaced with `return` and `mzero` respectively. The final refactored version of `inc` is in Figure 54.

It is worth saying that if the right hand side of the `Just` case from the original implementation of `inc` contained calls to `Nothing` then this function could not be generalised to `Monad`. This is because `Monad`s do not have a built in concept that can represent `Nothing`. The refactored function could be generalised to `MonadPlus` by replacing the `Nothing`s with `mzero`s.

Generalisation is a very common type of refactoring because it helps prevent repetition. A function with some specific behaviour can be generalised by extracting a sub-expression and passing that expression as a parameter instead. After the refactoring the target function's behaviour can be changed and applied in more places without having to repeat the implementation over again to change the extracted sub-expression. This section has discussed a refactoring to generalise a specific type, `Maybe`, to one of

```
inc :: Maybe Int -> Maybe Int
inc Nothing = Nothing
inc (Just i) = (Just (i + 1))
```

Figure 52: `inc`

```
inc :: Maybe Int -> Maybe Int
inc x = new_inc x f

new_inc :: Maybe Int -> (Int -> Maybe Int) -> Maybe Int
Nothing `new_inc` _ = Nothing
(Just i) `new_inc` f = f i

f i = Just (i + 1)
```

Figure 53: `inc` rewritten to look more like bind

its type classes, `MonadPlus` or `Monad`. Doing this generalises the target functions so that they can be applied to different values, rather than needing to implement separate functions for each specific type.

4.5 "List to Hughes List" Refactoring

The previous section discussed a refactoring that is most useful in the early stages of development when the details of data representation are in their infancy. As a project develops more and more decisions must be made about how data is represented and processed. Mid-development it may become necessary to change which data structures the program uses in order to facilitate adding additional features or for performance reasons.

This section covers a refactoring for automatically replacing a type by projecting that type into another (Erné et al. 1993). The two types (the original and new type) don't necessarily have an explicit relationship from the compiler's point of view (e.g. both implement the same type class) but instead present a semantically similar interface. For example, a binary search tree and a list support the same basic operations, such as

```
1 inc :: (Monad m) => m Int -> m Int
2 inc mi = mi >>= (\i -> (return (i+1)))
```

Figure 54: Final output from generalising `inc`

```
1 (++) :: [a] -> [a] -> [a]
2 [] ++ ys = ys
3 (x:xs) ++ ys = x:(xs ++ ys)
```

Figure 55: The standard definition of append

add, remove, and delete. Though the implementation details of lists and trees are very different, from the user's perspective all they need to know is what operations their chosen structure supports. The refactoring described in this section replaces one data type for another that supports similar operations.

4.5.1 Hughes Lists

Appending two lists into a single list is a fundamental operation over lists. The standard implementation of append is shown in Figure 55.

If the first argument to append is the empty list then append can just return the second argument. In the other case append traverses the first list popping off the head of the first list and recursively appending the tail of the first list and the second argument. The performance of append is therefore proportional to the length of its first argument. This has the unfortunate side effect that if a program builds a list by repeatedly appending a single element the end of a list, the program will spend significant amounts of time traversing the beginning of the list repeatedly. In total the performance of this function ends up being $O(n^2)$ where n is the length of the final list.

An example of a function that exhibits this behaviour is `countdown`, which is defined in Figure 56. `countdown` constructs a string that lists the numbers starting at the argument `i` down to zero. During each recursive call the previously computed result is traversed, but this parameter gets larger each time.

```
countdown :: Int -> String
countdown i = f i ""
  where f 0 s = s ++ "0"
        f i s = let newS = s ++ (show i) ++ ", " in
                  f (i-1) newS
```

Figure 56: The countdown function runs in $O(n^2)$ time.

```
> let lst = ([1,2,3] ++) . ([4,5,6] ++)
> :t lst
lst :: Num a => [a] -> [a]
> lst []
[1,2,3,4,5,6]
```

Figure 57: Building and deconstructing difference lists.

The poor performance of `countdown` quickly becomes noticable with `(countdown 10000)` taking around six seconds to run.[2]

Fortunately there is an alternative representative of lists that allows $O(n)$ time for such nested appends. This alternative representation was first described by John Hughes in (Hughes 1986) (hence "Hughes-lists"), they are also known as difference lists (O'Sullivan, Goerzen and Stewart 2008); and this is the nomenclature used in Hackage (Stewart and Leather 2017). In Hughes lists elements are stored as partial applications of the append function, these partial applications can then be composed using function composition (the `(.)` operator in Haskell) to append the two lists together.

Difference lists store the values `[1,2,3]` as `([1,2,3] ++)` which is of type `Num a => [a] -> [a]`. Appending `[4,5,6]` to `([1,2,3] ++)` first involves converting it to a difference lists (`([4,5,6] ++)` in this case) then these two difference lists can be appended with function composition which results in:
`([1,2,3] ++) . ([4,5,6] ++)`.

As seen in Figure 57, once it comes time to retrieve the normal list, the difference list is applied to an empty list. Function composition is applied from right to left so

[2] On an Intel i5 4690k processor with 16 GB of RAM

```
1 newtype DList a = DL {
2   unDL :: [a] -> [a]
3 }
4
5 fromList :: [a] -> DList a
6 fromList xs = DL (xs ++)
7
8 toList :: DList a -> [a]
9 toList (DL xs) = xs []
10
11 append :: DList a -> DList a -> DList a
12 append xs ys = DL (unDL xs . unDL ys)
```

Figure 58: The definition of `DList` taken from (O'Sullivan, Goerzen and Stewart 2008)

this keeps the left operand of `(++)` small. Internally difference lists are just a wrapper around a function from lists to lists. Figure 58 shows the definition of the `DList` new type which contains the partial application. The `unDL` function simply removes the `DL` constructor.

While difference lists support fast appends there is no such thing as a free lunch: speedups for certain functions are paid for by slowdowns in other places. Getting the head and tail of a normal list are both constant time operations but become linear time for difference lists because the difference lists will have to be converted back to normal lists.

4.5.2 Embeddable Types

This refactoring takes the view that a type consists of some structure and a set of functions that operate on that structure.

This refactoring requires that a source type is "reversibly embeddable" into the target type. If the source type is some type A and the target type is some type B then for A to be reversibly embeddable into B, two functions must exist: the projection function $proj :: A \to B$ and the abstraction function $abs :: B \to A$, as shown in Figure 59. The

$$abs.proj = id_A \tag{1}$$

apply	empty	singleton
cons	**snoc**	append
concat	replicate	**list**
head	tail	unfoldr
foldr	map	

Table 2: The DList API

property in equation 1 must hold for A to be embeddable in B. However, the reverse compositions of $proj.abs$ is not necessarily id_B.

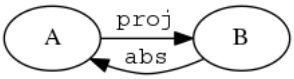


Figure 59: The relationship between the source type A and target type B and the respective projection and abstraction functions.

Intuitively this means that it needs to be possible to retrieve the A type from a B exactly as it was when it was projected into the new type. The reverse does not need to be true because any values of type B were introduced by the refactoring so any information lost converting a B back into an A wasn't in the original program and so does not need to be preserved to preserve behaviour.

In the "list to Hughes list" case the projection function is `fromList` (because it embeds lists into the new type `DList`) and the abstraction function is `toList`. How to define the set of pairs is an interesting problem with multiple solutions. The `Data.DList` module exports the functions shown in Table 2 (Stewart and Leather 2017).

From this list the two bold functions are the only functions without normal list counterparts. In the case of `snoc`, which appends a single element to the end of a `DList`, a normal list equivalent is not provided by the prelude because of the linear time performance of this operation. The `list` function, shown in Figure 60, makes up

```
list :: b -> (a -> DList a -> b) -> DList a -> b
list nill consit dl =
  case toList dl of
    [] -> nill
    (x : xs) -> consit x (fromList xs)
```

Figure 60: The definition of `list` from (Stewart and Leather 2017)

for the lack of pattern matching over `DLists`. The first two arguments to `list` are values that handle the two standard pattern match cases over lists, an empty list, and when the list contains at least a single element. The third argument to `list`, "`dl`", is the difference list that pattern matching should be performed on. If `dl` is empty than `list`'s first argument "`nill`" is returned. In the other case `list` will pass the head and the tail of `dl` to it's second argument a function that performs the computation when the list is non-empty.

The rest of the `DList` API could be paired with equivalent list functions. However, it's not necessarily a good idea for every normal list function to be refactored to its `DList` equivalent. As was mentioned earlier, certain `DList` functions are less efficient than the corresponding normal function, so the primary purpose of this refactoring is defeated if the refactored code runs slower than the original source.

Fortunately different versions of the refactoring can be made by defining separate sets depending on the behaviour that is desired. For example one set could only include the `DList` constant time operations (`append`, `empty`, and `cons`) and another set could include all possible pairings of operations in Table 2.

Which functions should be changed is highly dependent on the relationship between the source and target types, the motivation for the refactoring, and how the source program is constructed. The list to Hughes replacement is motivated for performance reasons so limiting the replacements to only operations that run in constant time makes sense. If the refactoring is being used to completely upgrade a system from using one type to another then the programmer will want to make sure that every function used

by the source type can be replaced. The implementation of this refactoring in HaRe is as adaptable as possible so that users can customize it to meet their specific needs (see Chapter 5).

4.5.3 Refactoring functions to use Hughes lists

For functional programmers, lists are a very familiar and versatile data structure. However, if an application requires repeated appends as described at the start of this section their performance becomes an issue. Difference lists provide a similar interface to lists but without this troublesome behaviour and provide projection (`fromList`) and abstraction (`toList`) functions between themselves and normal lists. This section will describe a refactoring to convert programs written using normal lists to use difference lists instead.

The refactoring is different depending on which types of the target function are lists. There are three different cases:

1. A list is the type of parameters only
2. A list is the result type
3. Both the result type and at least one parameter are lists

Each of these cases differs in the way that the Hughes list type is introduced to the function. In the first case the type changes begin at the leaves and the refactoring needs to fix any type errors from the bottom towards the top of the tree. When the result type is the only list instance in the type signature the opposite process must happen, the new type needs to be “pushed” down the AST from the root. In the final case the refactoring will attempt to convert the entire AST to use Hughes lists.

Though the general scheme the refactoring takes to transform the target function is simple to understand, how the refactoring actually goes about transforming is more challenging. This is because for any given function there are multiple possible “correct"

```
insComma :: String -> String -> String
insComma s1 s2 = s1 ++ "," ++ s2
```

Figure 61: `insComma`

```
insComma_1 :: DList Char -> DList Char -> DList Char
insComma_1 s1 s2 = s1 `append` fromList (",") `append` s2

insComma_2 :: DList Char -> DList Char -> DList Char
insComma_2 s1 s2 = fromList ((toList s1) ++ (",") ++ (toList
    s2))
```

Figure 62: Two possible refactorings for `insComma`

definitions the refactoring could produce. Take for example the function `insComma` in Figure 61.

When refactoring both of the arguments and the result type of `insComma` to become `DList Char` there are multiple ways to refactor this function. Figure 62 shows two possibilities; which one should the refactoring produce and why?

To make this decision, the refactoring is designed to reflect the assumption that because this function is being refactored to use the new type, the user wants the new type to be used in as many places as possible. This makes the first definition preferable to the second one because it only converts a single item using `fromList` and replaces the appends, whereas the second example converts the two arguments into lists, appends everything together and then converts the result back into a `DList`. The refactoring prioritises minimising the number of conversion introduced into the refactored program.

4.5.4 Modifying the Type of an Input

The simplest case of this refactoring is modifying the type of a parameter. Consider the example in Figure 63 that calculates the mean of list of numbers.

Refactoring `means`' first argument to become a Hughes list is fairly straight forward matter of wrapping the, now of type `DList`, parameter `lst` with the abstraction

```
1 mean :: Fractional a => [a] -> a
2 mean lst = foldr (+) 0 lst / length lst
```

Figure 63: Calculating the mean of a list.

```
1 mean :: Fractional a => DList a -> a
2 mean lst = foldr (+) 0 (DL.toList lst) / length (DL.toList lst
      )
```

Figure 64: `mean` refactored

function `toList`. The refactored version of `mean` is in Figure 64. The functions from the `DList` library have been prefaced with the "`DL`" qualifier, to differentiate which functions come from which API. This convention will be used through the remainder of the section. You'll also notice that `toList lst` is calculated twice; this is a good candidate for further refactoring by extracting this expression into a `let` or a `where` clause.

This example is one of the cases where there are multiple possible refactorings. Figure 65 shows a different version of a refactored `mean`. Because `foldr` is defined both for difference lists and normal lists, the abstraction function could be added around that expression instead. Which version should the refactoring produce?

In this case, where the refactoring targets a parameter and the result is a type other than a Hughes list, the refactoring assumes that the programmer wants the Hughes list converted as soon as possible in the function so the `fromList` conversions will be wrapped around the parameters.

4.5.5 Modifying the Result Type

The second example covers the case where the result type of a function is changed to `DList` instead of list. Figure 66 contains the definition of a simple algebraic data type of a tree, a function (`enumerate`) that returns an in-order list of all the tree's elements, and a function that prints a tree's enumeration to standard output.

```
mean :: Fractional a => DList a -> a
mean lst = (DL.foldr (+) 0 lst) / length (DL.toList lst)
```

Figure 65: `mean` refactored another way.

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

enumerate :: Tree a -> [a]
enumerate Leaf = []
enumerate (Node left x right) = (enumerate left) ++ [x] ++ (
    enumerate right)

printEnumTree :: (Show a) => Tree a -> IO ()
printEnumTree tree = let lst = enumerate tree in
  print lst
```

Figure 66: Definition of enumerate

This time the refactoring will affect the result type of the function. As opposed to the previous example where changes occurred at the leaves of the abstract syntax tree this case of the refactoring changes the type of the AST's root.

Changing the type of the AST's root requires the refactoring to traverse the tree top down from left to right. This is because the result type of any syntax tree is determined by the function (or value in the case of tree with only a single node) in the leftmost child.

`enumerate`'s first case is simple enough to refactor. There is only a single value in the tree the empty list literal. This node's current type is `[a]` and it needs to become `DList a`. When the refactoring reaches the `[]` value it searches to see if it is paired with some difference list operation in the set of pairs. The empty list literal is paired with the difference list operation `empty`, the refactoring sees this and replaces the empty list with `DL.empty :: DList a`.

After the replacement of `[]`, `enumerate`'s first case is successfully refactored. The second case of `enumerate` is more complex than the first. Figure 67 shows the

syntax tree for `enumerate`'s second case.

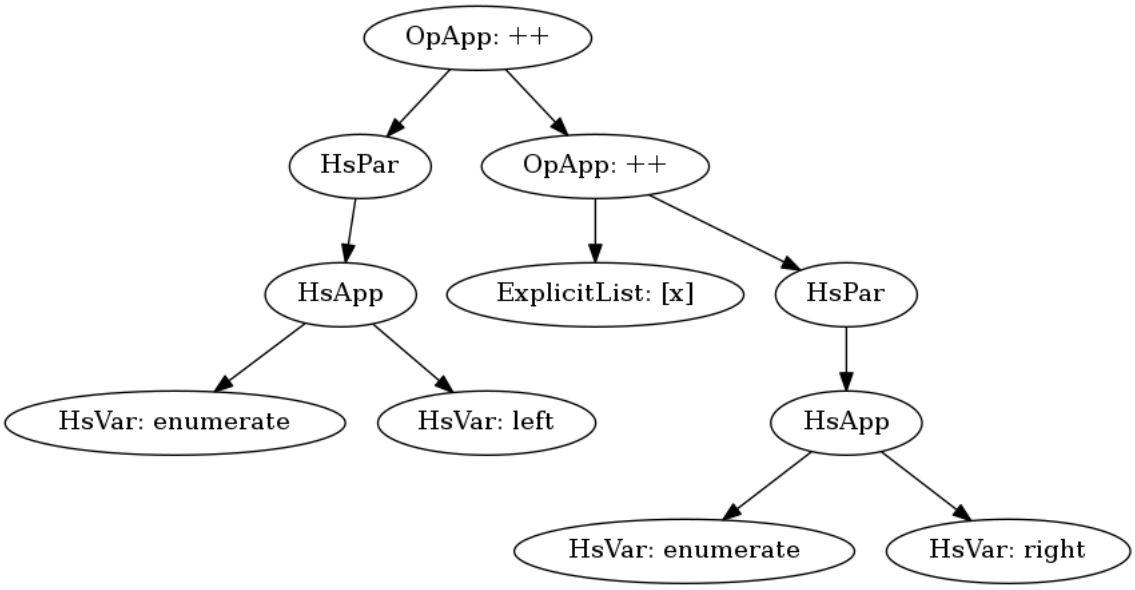


Figure 67: A simplified syntax tree of `enumerate`'s second case.

The refactoring begins the traversal at the top of the syntax tree with the goal of modifying the entire tree to have a result type of `DList a`. The root node of the tree is the operator application of the left-hand append operation. The result type of this instance of append is the type that determines the whole tree's result type. The standard append operation is paired with the difference list `DL.append` operation. The refactoring checks to ensure that the difference list append has the correct result type and, because it does, makes the replacement.[3] After replacing `++` with `DL.append`, the result type is correct but the function will no longer type check because `++` and `DL.append`'s arguments are not the same types.

The refactoring then must recurse down both the left and right subtrees to change their types from `[a]` (the type of `(++)`'s arguments) to `DList a` (the type `append`'s arguments).

The left subtree of the root node is shown in Figure 68. The refactoring can descend

[3]The refactoring will automatically make `DL.append` infix by surrounding the call with backtick characters since this replacement is modifying an operator application.

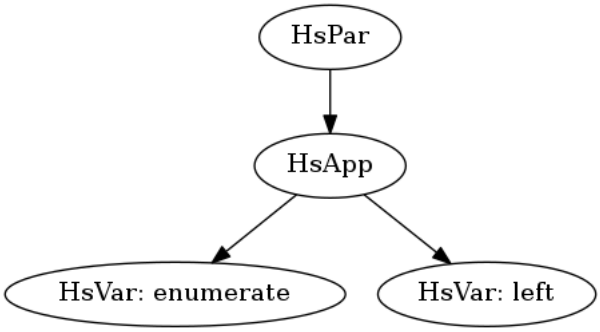


Figure 68: The left subtree of `enumerate`'s second case.

through the parentheses (represented by the `HsPar` constructor); the refactoring continues down the left side of the function application (`HsApp`). When the refactoring encounters the call to `enumerate` it recognises that this is the recursive call and even though the type of `enumerate` stored in the syntax tree is `Tree a -> [a]` after the refactoring its type will be `Tree a -> DList a` which is the correct result type for this subtree. The refactoring can now confirm that the left subtree is of type `DList a` without checking the right side of the application because the refactoring didn't change the type of `enumerate`'s arguments.

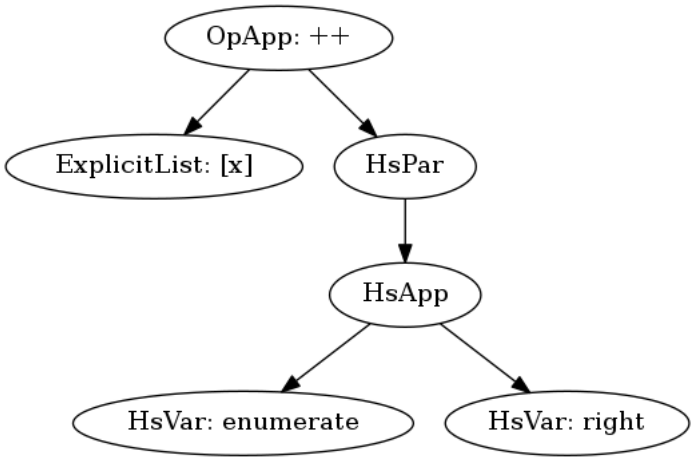


Figure 69: The right subtree of `enumerate`'s second case.

```
1 enumerate :: Tree a -> DList a
2 enumerate Leaf = empty
3 enumerate (Node left x right) = (enumerate left) `append` (
    singleton x) `append` (enumerate right)
```

Figure 70: The refactored definition of `enumerate`

After refactoring the left subtree, the right subtree needs to be modified to be of type `DList a` as well. Shown in Figure 69, the right subtree's root is a second call to `(++)`. Once again the refactoring replaces `++` with `DL.append` because `append`'s result type is also `DList a`. Doing this replacement sets off additional traversals that are designed to ensure that the two arguments to the root node become typed `DList a` as well. This is possible because the original program is well-typed and the arguments to append are of type list so at the very least the arguments can be converted to `DLists` with `fromList`. The right argument to the root of this subtree is the `(enumerate right)` expression which is handled in the same way the `(enumerate left)` call was handled. The left subtree of Figure 69 is the list literal `[x]`. This can be replaced with a call to `(DL.singleton :: a -> DList a)` the equivalent difference list function. At this point the refactoring is finished modifying the definition of `enumerate` and the result can be seen in listing 70.

The refactoring is not finished yet, however. The original definition contained in Figure 66 had another function `printEnumTree` that depended on `enumerate`. The final modification that needs to happen to this example is to wrap all calls to `enumerate` in *non-refactored* definitions with the abstraction function to convert the result back to a list. The final product of the refactoring can be seen in Figure 71.

4.5.6 Modifying Parameter and Result Types

The final case of this refactoring involves modifying both the result type and one (or more) of the parameters of the target function. This example will use the `explode` function from Figure 73 and will refactor both its argument and result type to become

```
1  data Tree a = Leaf
2           | Node (Tree a) a (Tree a)
3
4  enumerate :: Tree a -> DList a
5  enumerate Leaf = empty
6  enumerate (Node left x right) = (enumerate left) `append` (
       singleton x) `append` (enumerate right)
7
8  printEnumTree :: (Show a) => Tree a -> IO ()
9  printEnumTree tree = let lst = toList (enumerate tree) in
10   print lst
```

Figure 71: The final product of the refactoring

```
1  explode :: [a] -> [a]
2  explode lst = concat (map (\x -> replicate (length lst) x) lst
       )
```

Figure 72: The initial definition of `explode`

`DList a`. `explode` is a silly function that replicates each of its elements a number of times equal to the length of the list, for example, `explode [1,2,3]` returns `[1,1,1,2,2,2,3,3,3]`. This is an odd function but it has two characteristics we are interested in, (a), both its parameter and result type are of type list and, (b), it uses the `map` higher ordered function which is an interesting case for this refactoring. The abstract syntax tree of `explode` is in Figure 73.

Much like the previous case where just the result type was modified, the refactoring will start working on the syntax tree in a top-down manner modifying the function definition to have the correct result type. This case could also be implemented in a bottom-up manner but the top-down method allows for more code reuse from the implementation of the case when the result type is modified.

The refactoring starts on the left subtree with the call to:

`concat :: Foldable t => t [a] -> [a].`

The equivalent difference list function is:

`DL.concat :: Foldable t => t (DList a) -> DList a`

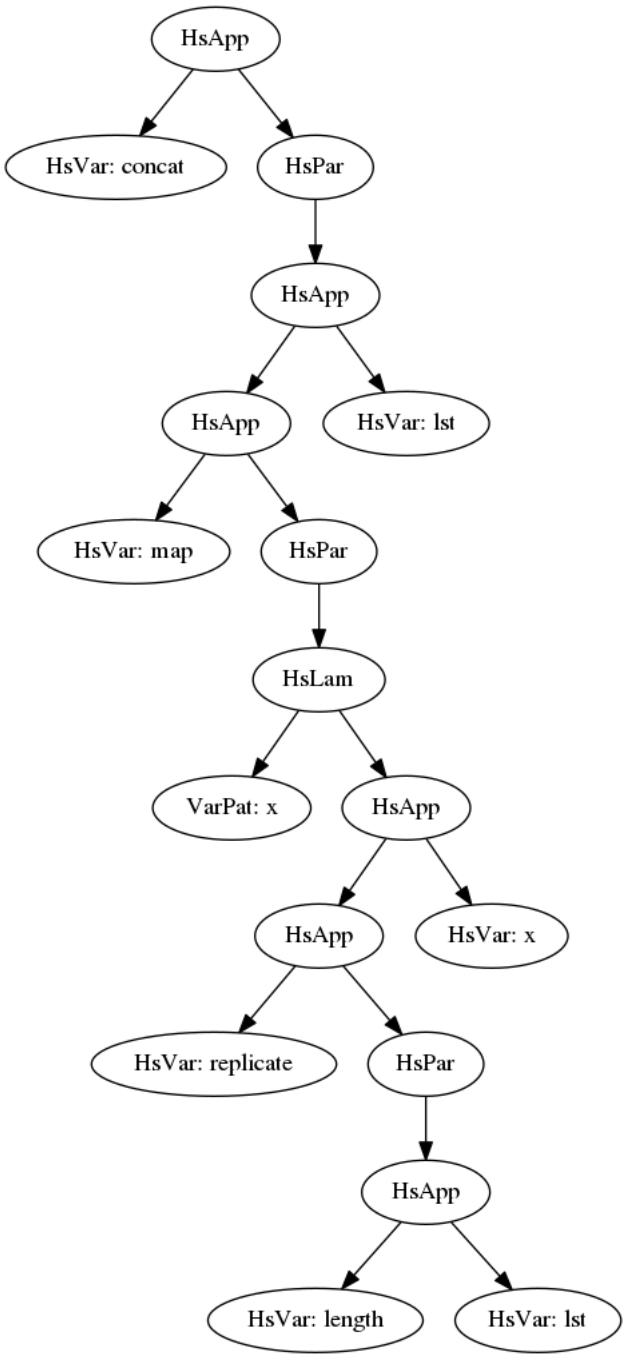


Figure 73: A simplified representation of `explodes`'s syntax tree

Since the result type of this version of `DL.concat` is `DList a` the refactoring performs the switch between the two functions.

After the change on the left subtree, the refactoring needs to modify the right subtree so that it is of type `Foldable t => t (DList a)` rather than its current type of `Foldable t => t [a]`. The leftmost child of the right subtree is the call to `map :: (a -> b) -> [a] -> [b]`. The difference list equivalent map is appropriately typed `DL.map :: (a -> b) -> DList a -> DList b`. Should the refactoring change this node to the difference list version? And if so what changes will be need to made to other subtrees?

The refactoring will swap this node out if `Foldable t => t (DList a)` (the type of `DL.concat`'s parameter) can be the same type as `DList b` (`DL.map`'s result type); since `DList` is a member of the `Foldable` type class, the swap can happen as long as the `b` type variable in `map`'s type is equal to `DList a`. This node's type after the swap and filling in the known type variables is:

```
DL.map :: (a -> DList b) -> DList a -> DList (DList b)
```

This new type's arguments' types are both different from the original types so the refactoring will need to check both of these subtrees as well.

Starting with the lambda expression (the syntax tree in Figure 74) the refactoring needs to modify this expression's type to become `a -> DList b`[4]. The current type of the lambda expression is `a -> [a]` so the refactoring only needs to modify its result type. The refactoring can then proceed to the left-most child of this expression, the call to `replicate :: Int -> a -> [a]`. The refactoring swaps this call for the difference list version of `DL.replicate :: Int -> a -> DList a`.

If this refactoring were only modifying the result type of `explode`, the refactoring would be done working with the lambda expression because the type change affects the syntax tree in a top down manner. In this case, however, because the type of `explode`'s argument was also changed, so leaves of the syntax tree can also have changed type. This means that every subtree needs to be checked to ensure that it still type checks.

[4] `a` and `b` could be the same type.

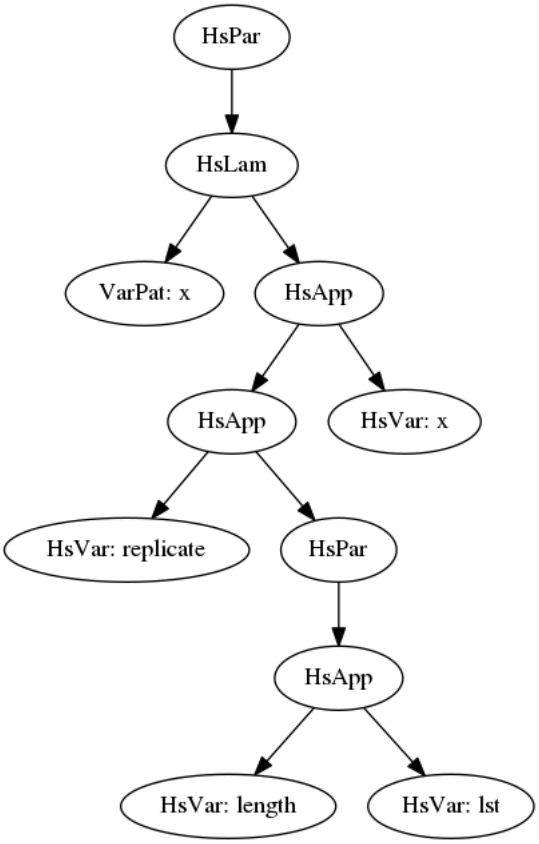


Figure 74: The syntax tree of the lambda expression in `explode`.

The untouched sections of this expression's syntax tree are `replicate`'s arguments. The second argument is just the variable `x` which hasn't changed type. The first argument on the other hand is the expression `(length lst)` and `lst` is the argument of `explode` that is now a difference list rather than a normal list. Ideally the refactoring would rewrite this expression by replacing the normal list functions (`length` in this case) with difference list equivalents. Unfortunately there is no difference list version of `length` so the refactoring will have to introduce the abstraction function to convert back to a normal list to calculate its length.

The lambda expression has now been successfully refactored. The rewritten lambda expression is in Figure 75.[5]

[5]To help clarify which functions are for lists and which are the difference list versions, all difference

```
1 (\x -> DL.replicate (length (DL.toList lst)) x)
```

Figure 75: The refactored lambda expression.

```
1 import qualified Data.DList as DL
2 import Data.DList (DList)
3
4 explode :: DList a -> DList a
5 explode lst = DL.concat
6   (DL.map
7     (\x -> DL.replicate (length (DL.toList lst)) x)
8     lst)
```

Figure 76: The final refactored result of `explode`.

Once refactoring the lambda expression is finished there is only a single node of `explode` that the refactoring hasn't touched yet, the use of `lst` as the second argument of `map`. When the refactoring swapped `map` for its difference list version it was able to determine the new type of both its arguments, `(a -> DList b)` for the first argument (the lambda expression) and `DList a` (`lst`). Fortunately because `lst` is a target of the refactoring, its new type is `DList a` so this node can remain unchanged.

The final result of this case of the Hughes list refactoring is in listing 76.

4.6 Summary

This chapter has introduced the concept of data-driven refactorings for the object-oriented programming paradigm and described three of the data-driven refactorings developed for this thesis. The first, "introducing a type signature", creates an additional abstraction to better describe what types are representing. "Generalising Maybe" takes code written for a specific type and generalises it so that it is applicable to more types. Finally the "list to Hughes list" refactoring replaces one type with another equivalent type. This refactoring allows a project to be retyped mid-development because it is not

list functions have been qualified with the `DL` identifier.

always clear at the beginning of a project what the correct data representation should be.

The following chapters will expand on some of these ideas. The next chapter will describe another form of generalisation that rewrites monadic code into its equivalent applicative functor code. The applicative code can be much cleaner and descriptive in certain cases. It can also allow programmers to take advantage of a different way of executing their programs. Chapter 7 discusses introducing effectful abstractions into pure code. This process helps to add additional features to a code base, such as shared state, mid-development.

Chapter 5

Implementing Data-Driven Refactorings in HaRe

The previous chapter described three refactorings, "introducing a type synonym," "generalising `Maybe`," and the "list to Hughes list refactorings". This chapter will describe how refactorings are implemented in HaRe. This chapter is organised in a similar way to the previous one. Beginning with Section 5.1 which describes the implementation of the generalising `Maybe` refactoring that was discussed in Section 4.4. Section 5.2 covers the implementation of the "list to Hughes list" refactoring (see Section 4.5) and the embeddable type refactoring system that it is made with. In addition to discussing the implementation of the refactorings from Chapter 4 this chapter also covers the enhancements that HaRe's API has undergone while developing the refactorings discussed in this thesis. Section 5.3 discusses the enhancements I made to HaRe's API.

5.1 Implementation of the "Generalising Maybe" refactoring

The design of the generalising `Maybe` refactoring was discussed in the previous chapter (see Section 4.4). This section will describe its implementation in the HaRe refactoring

tool.

This refactoring tries to generalise something of type `Maybe` to become either of type `Monad` or `MonadPlus`. The implementation of this refactoring attempts to produce the “most general" version of the source program. This means that when possible the refactoring will replace the targetted `Maybe` type with a value of type `Monad`; if this is not possible the value will be of type `MonadPlus`, and finally if that also is not possible the refactoring cannot continue and the source and target programs of the refactoring are identical.

The implementation of the “Generalising Maybe” refactoring in HaRe attempts to convert the `Maybe` types in a function to the “most general” type class possible. This means that if the `Maybe` type can be replaced with `Monad` it will be, otherwise the function will be rewritten using `MonadPlus` instead. First the refactoring will replace all of the `Maybe` constructors in right hand side expressions with the corresponding `MonadPlus` operation (`Just` becomes `return` and `Nothing` becomes `mempty`). At this point the refactoring checks if the target function’s computations were done entirely inside of `Maybe`, that is the function receives a `Maybe` value as a parameter, returns a value of type `Maybe`, and handles the `Nothing` constructor by just returning `Nothing`; when all of these things are true the target function is really just replicating the functionality of `Maybe`’s bind instance. In this case the refactoring rewrites the function body to use bind instead. Finally depending on how generalisable the target function was, the refactoring will update the type signature by replacing the `Maybe`s with “`m`” type variables that are bound the appropriate type class, either `MonadPlus` or `Monad`.

This refactoring, at the top level (defined in Figure 77), determines which type the target program can be generalised to. This is determined by whether the target function contains a “`Nothing` to `Nothing`" case. When this is true then the function can be rewritten with the `Monad` interface. This check is defined in the `containsNothingToNothing` function, which traverses the AST of the target

```
1  doMaybeToPlus :: FilePath -> SimpPos -> String -> Int ->
       RefactGhc ()
2  doMaybeToPlus fileName pos@(row,col) funNm argNum = do
3    parsed <- getRefactParsed
4    let mBind = getHsBind pos parsed
5    case mBind of
6     Nothing -> error "Function bind not found"
7     Just funBind -> do
8        hasNtoN <- containsNothingToNothing funNm argNum pos
             funBind
9        case hasNtoN of
10         True -> do
11           doRewriteAsBind fileName pos funNm
12         False -> do
13           canReplaceConstructors <- isOutputType funNm argNum
                pos funBind
14           case canReplaceConstructors of
15             True -> do
16               logm $ "Can replace constructors"
17               replaceConstructors pos funNm argNum
18             False -> return ()
19       return ()
```

Figure 77: The top level function of the generalising `Maybe` refactoring.

function searching for a case where the `Pat` on the left hand side pattern is only `Nothing` and the `HsExpr` on the right hand side of the match is only `Nothing`. If this is true for the target function then `doRewriteAsBind` modifies the other case(s) of the function.

The other case has the refactoring checking if its possible to replace all of `Maybe`'s constructors with the equivalent `MonadPlus` values. This is checked in `isOutputType` which checks whether the target type of the refactoring is also the result type of the function. When this is the case `MonadPlus`' constructors can be substituted for `Maybe`'s `Nothing` and `Just`. This is done by the `replaceConstructors` function.

The rest of this section will discuss the two functions that modify the source function, `doRewriteAsBind` and `replaceConstructors`. There will also be a brief

```
1  doRewriteAsBind :: FilePath -> SimpPos -> String -> RefactGhc
       ()
2  doRewriteAsBind fileName pos funNm = do
3    parsed <- getRefactParsed
4    let bind = gfromJust "doRewriteAsBind" $ getHsBind pos parsed
5        matches = GHC.mg_alts . GHC.fun_matches $ bind
6    if (length matches) > 1
7      then error "Multiple matches not supported"
8      else do
9      let (GHC.L _ match) = head matches
10     (varPat, rhs) <- getPatAndRHS match
11     (newPat, _) <- liftT $ cloneT varPat
12     (newRhs, _) <- liftT $ cloneT rhs
13     let rhs = justToReturn newRhs
14     lam <- wrapInLambda newPat rhs
15     let newNm = case newPat of
16                   (GHC.L _ (GHC.VarPat nm)) -> mkNewNm nm
17                   _ -> mkRdrName $ "m_value_" ++ funNm
18     new_rhs <- createBindGRHS newNm lam
19     replaceGRHS funNm new_rhs newNm
20     fixType funNm
21       where mkNewNm rdr = let str = GHC.occNameString $ GHC.
           rdrNameOcc rdr in
22             GHC.Unqual $ GHC.mkVarOcc ("m_" ++ str)
```

Figure 78: The implementation of `doRewriteAsBind`. The function that generalises `Maybe` to `Monad`

section on enhancements that can be made to this implementation of the generalise `Maybe` refactoring to make it more robust.

5.1.1 Generalising to `Monad`

Figure 78 shows the implementation of `doRewriteAsBind` the function that rewrites a function of type `Maybe` to use the monadic bind interface instead. The first three lines of `doRewriteAsBind` (lines 3-6 in Figure 78) extract the list of matches that make up every Haskell function. A match consists of the left hand side pattern binding and the right hand side expression. For example, in Figure 79, the `last` function's AST representation consists of three matches, one for each function body, whereas the AST

```
1 last :: [a] -> Maybe a
2 last [] = Nothing
3 last [x] = Just x
4 last (x:xs) = last xs
```

Figure 79: The AST of the `last` function contains three matches.

of `doRewriteAsBind` only has one match because its definition contains only a single body.

Once the target function's list of matches is retrieved `doRewriteAsBind` can then perform the appropriate refactoring for a target function with a single match or multiple matches[1]. Currently this implementation does not support multiple matches but the refactoring could be expanded to automatially wrap the target function's pattern matches in a `case` statement. Another way to allow HaRe to support this case is to create a separate refactoring that rewrites functions with multiple bindings to use a `case` statement instead, which would allow the refactoring to proceed.

After checking that there is only a single match in the AST of the target function, `doRewriteAsBind` can start modifying that function. The rest of `doRewriteAsBind` (from line 10 in Figure 78) constructs a lambda expression and variable to make up the new right hand side of the target function.

This is done by taking the right hand side expression from the target function's match and replacing any calls to `Just` with `return`. This generalised version of the right hand side expression can then be wrapped in a lamdba expression with the original pattern being the binding for this expression (line 14's call to `wrapInLambda`). This new lambda expression can be bound to the newly created variable `newNm`. This new variable keeps the name given to the original pattern if it was just a simple name (e.g. "`x`") and appends "`m_`" to this name to prevent a naming conflict with the variable inside of the lambda expression. In the case that the pattern was more complex, like

[1] It's important to note that `containsNothingToNothing`, the function that checks if the target function has a `Nothing` to `Nothing` case, also removes this case from the AST. This means that the `matches` list excludes this case.

when matching a data type constructor (e.g. `Just x`) or a list `(x:xs)`, then a generic variable name is created using the target function's name.[2]

The new name is then bound to the lambda expression (line 18) and the target function's body is replaced with this bind expression. The step of the rewriting generalises the type signature to be use the `Monad` type class in the place of `Maybe`.

5.1.2 Generalising to `MonadPlus`

When the target function cannot be generalised to `Monad` because it does not contain the `Nothing` to `Nothing` case, it may be possible to generalise it to use the `MonadPlus` type class instead. This section will cover how this case of the refactoring is implemented in HaRe.

An instance of `MonadPlus` is a `Monad` with a monoidal structure. This structure is defined by an associative operator `mplus` and its identity value `mzero` (Yorgey 2009b). In `Maybe`'s case `mzero` is defined as `Nothing` and `Maybe`'s other constructor, `Just`, is the value of `Maybe`'s definition `return`. This case of the refactoring just replaces instances of `Just` with `return` and `Nothing` with `mzero`.

Figure 80 shows the top level function `replaceConstructors` that modifies a target function to replace the `Maybe` specific values with `mzero` and `return`. This function primarily consists of two traversal patterns, the "stop top down traversal" (`stop_tdTP`) in `applyInGRHSs` and the `everywhereM` in `runGRHSFun`. The only parts of the function that need to be changed are in the right hand side of the target function; the first traversal (from `applyInGRHSs`) descends to this level and then stops when the guarded right hand side type is found. From there the other traversal continues down the tree replacing the instances `Just` and `Nothing`.

After the right hand side binding has been rewritten, the target function's original binding is replaced with the new binding in the call to `replaceBind` on line six

[2]The user of the refactoring will most likely want to apply the renaming refactoring to this variable to give it a more appropriate name.

```
1  replaceConstructors :: SimpPos -> String -> Int -> RefactGhc
       ()
2  replaceConstructors pos funNm argNum = do
3   parsed <- getRefactParsed
4   let (Just bind) = getHsBind pos parsed
5   newBind <- applyInGRHSs bind replaceNothingAndJust
6   replaceBind pos newBind
7   fixType' funNm argNum
8    where applyInGRHSs :: (Data a) => UnlocParsedHsBind -> (a
        -> RefactGhc a) -> RefactGhc UnlocParsedHsBind
9         applyInGRHSs parsed fun = applyTP (stop_tdTP (failTP `
            adhocTP` (runGRHSFun fun))) parsed
10        runGRHSFun :: (Data a) => (a -> RefactGhc a) ->
            ParsedGRHSs -> RefactGhc ParsedGRHSs
11        runGRHSFun fun grhss@(GHC.GRHSs _ _) = SYB.everywhereM
            (SYB.mkM fun) grhss
12        mzeroOcc = GHC.mkVarOcc "mzero"
13        nothingOcc = GHC.mkVarOcc "Nothing"
14        returnOcc = GHC.mkVarOcc "return"
15        justOcc = GHC.mkVarOcc "Just"
16        replaceNothingAndJust :: GHC.OccName -> RefactGhc GHC.
            OccName
17        replaceNothingAndJust nm
18          | (GHC.occNameString nm) == "Nothing" = do
19             logm "Replacing nothing"
20             return mzeroOcc
21          | (GHC.occNameString nm) == "Just" = do
22             logm "Replace just"
23             return returnOcc
24          | otherwise = return nm
```

Figure 80: The `replaceConstructors` function replaces `Maybe`'s constructors with more general values.

of Figure 80. Replacing the original binding of a function with a refactoring one is a common operation that many refactorings need to perform. Due to its re-usability `replaceBind` is a function from HaRe's API and will be discussed in section 5.3.

The final step of this refactoring changes the type signature of the target function. The instances of `Maybe` in the type signature need to be replaced with a type variable that is bound to the `MonadPlus` typeclass. The `fixType'` function performs this

```
1  fixType' :: String -> Int -> RefactGhc ()
2  fixType' funNm argPos = do
3    logm "Fixing type"
4    parsed <- getRefactParsed
5    let m_sig = getSigD funNm parsed
6        (GHC.L sigL (GHC.SigD sig)) = gfromJust "fixType'" m_sig
7    fixedClass <- fixTypeClass sig
8    replacedMaybe <- replaceMaybeWithVariable fixedClass
9    newSig <- locate (GHC.SigD replacedMaybe)
10   addNewKeyword ((G GHC.AnnDcolon), DP (0,1)) newSig
11   synthesizeAnns newSig
12   addNewLines 2 newSig
13   newParsed <- replaceAtLocation sigL newSig
14   anns <- liftT getAnnsT
15   putRefactParsed newParsed anns
```

Figure 81: `fixType'` is a function that fixes the type signature of a function that is being generalised from `Maybe` to `MonadPlus`.

rewriting. The definition of `fixType'` is shown in Figure 81.

The definition of `fixType'` is shown in Figure 81, this functions follows the following steps:

- *Line 5:* Retrieve the signature from the parsed abstract syntax
- *Line 7:* Insert the binding of the variable "`m`" to the `MonadPlus` typeclass
- *Line 8:* Replace the instances of `Maybe` with the "`m`" type variable
- *Lines 9 - 12:* Create the appropriate annotations for the new syntax elements
- *Line 13:* Replaces the old type signature in the parsed source
- *Line 15:* Updates the refactoring state with the modified parsed source

The `fixTypeClass` and `replaceMaybeWithVariable` functions transform the abstract syntax of the type signature, and the rest of `fixType'` creates and modifies the associated annotations so that all of its elements appear in the target program

and are well formatted. For example, the call to `addNewKeyword` on line 10 of Figure 81 associates the `AnnDcolon` annotation with the new type signature. This annotation represents the "`::`" operator that is otherwise not represented explicitly in the abstract syntax tree. These annotations are what is used by ghc-exactprint to determine where syntax elements that are not located by the abstract syntax are and how the target module should be formatted. This was discussed in more detail in section 3.4.

This section has covered the implementation of the generalising `Maybe` refactoring. This is a fairly straightforward refactoring implementation, the preconditions are checked up front and depending on the structure of the target function the correct transformation is chosen. The next section will describe the implementation of the "list to Hughes list" refactoring. This refactoring must descend the entire syntax tree of each target function and transformations may occur based on the type of the AST's subtrees.

5.2 Implementation of the "List to Hughes List" refactoring

The "list to Hughes list" refactoring was first described in section 4.5. This section will describe the implementation of that refactoring in HaRe. Section 4.5 also introduced the concept of a type that can be "embedded" into another. This means that the source type, which in this particular case is `[a]`, can be transformed into the target type (`DList a`) and the original value of the object can be retrieved from the target type.

Though "list to Hughes list" is the specific refactoring outlined in this thesis and implemented in HaRe, this sort of refactoring can be applied to any pair of types that hold the properties described in section 4.5. HaRe's implementation of the "list to Hughes list" refactoring keeps this in mind and efforts were made to build reusable components so that similar refactorings can be easily implemented for different types.

The `doHughesList` function shown in Figure 82 is main function that performs the "list to Hughes list" refactoring. This function performs four primary tasks, adding

```
1  doHughesList :: FilePath -> String -> SimpPos -> Int ->
       EmbFuncStrings -> RefactGhc ()
2  doHughesList fileName funNm pos argNum fStrs = do
3    let mqual = Just "DList"
4    addSimpleImportDecl "Data.DList" mqual
5    ty <- getDListTy mqual
6    parsed <- getRefactParsed
7    let
8      (Just lrdr) = locToRdrName pos parsed
9      rdr = GHC.unLoc lrdr
10     dlistCon = getTyCon ty
11     newFType = resultTypeToDList dlistCon
12     (Just funBind) = getHsBind rdr parsed
13     (Just tySig) = getTypeSig pos funNm parsed
14     newResTy = getResultType ty
15   iDecl <- dlistImportDecl mqual
16   iSt <- getInitState iDecl fStrs "toList" "fromList" mqual
       newResTy
17   bind' <- embRefact argNum mqual rdr ty iSt funBind
18   replaceFunBind pos bind'
19   newTySig <- fixTypeSig argNum tySig
20   replaceTypeSig pos newTySig
21   let modQual = case mqual of
22                   (Just s) -> s ++ "."
23                  Nothing -> ""
24   fixClientFunctions modQual (numTypesOfBind funBind) argNum
       rdr
25   addConstructorImport
```

Figure 82: `doHughesList` is the top level function of the Hughes list refactoring.

the import declaration for the difference list library, transforming the function definition, replacing the existing function definition, and wrapping the call points of the target function with the abstraction function.

The most interesting part of this refactoring is the transformation of the function definition. The other steps are handled by HaRe's API which is discussed in section 5.3. The transformation occurs on lines 16 and 17 of Figure 82. The transformation runs within its own state and line 16 creates the initial value of this state while line 17 actually runs the stateful computation. The state's type is shown in Figure 83.

```
1  data EmbRefactState = EmbState {
2    funcs :: EmbeddableFuncs,
3    typeStack :: [Maybe GHC.Type],
4    insertAbs :: Bool
5                            }
6
7  data EmbeddableFuncs = IF {
8    projFun :: GHC.RdrName,
9    absFun :: GHC.RdrName,
10   eqFuns :: M.Map String (GHC.RdrName, GHC.Type)
11   }
```

Figure 83: The type of the state that the refactoring runs in.

.

The state consists of three fields, `funcs` keeps track of the functions that operate over the source type and their equivalent functions that operate on the target type, and the names of the projection and abstraction functions. The `typeStack` field is a stack that keeps track of what changes need to be made further down the abstract syntax tree. The type stack is described in more detail in section 5.2.1 below. Finally, the `insertAbs` field keeps track of when the abstraction function needs to be applied to certain values.

The call to `getInitState` on line 16 of Figure 82 takes in the import declaration of the target type, a list of pairs of strings that represent the equivalent functions over the source and target types, the names of the projection and abstraction functions, the target type's import qualifier if it exists, and the final result type of the target function. In the "list to Hughes list" case the import declaration is "`import Data.DList qualified as DList`." Because many of the Hughes list functions have the same names as their normal list counterparts a qualifier is required so `DList` is chosen.

The list of pairs associates the normal list functions with the appropriate Hughes list functions. A single pair in this list contains, first, the normal list function name followed by the Hughes list function that performs the same operation. For example, "`(:)`" is paired with its Hughes list equivalent "`cons`" because both functions add a

```
1 [("[]","empty"),(":","cons"),("++","append"),("concat", "
    concat"),("replicate","replicate"), ("head","head"),("tail"
    ,"tail"),("foldr","foldr"),("map","map"), ("unfoldr", "
    unfoldr")]
```

Figure 84: The list of associated function pairs for the "list to Hughes list" refactoring

```
1 embRefact :: Int -> Maybe String -> GHC.RdrName -> GHC.Type ->
     EmbRefactState -> ParsedBind -> RefactGhc ParsedBind
2 embRefact _ mqual funNm newFTy iST bnd = modMGAltsRHS (\e ->
    runEmbRefact (doEmbRefact e) iST) bnd
```

Figure 85: The definition of `embRefact`

single element to the beginning of a list. The full list of pairs for the "list to Hughes list" refactoring is shown in Figure 84.

The projection and abstraction functions are "`toList`" and "`fromList`" respectively in the "list to Hughes list" case. The final result type of the target function depends on the result type of the source function. If the result type of the source function is `[Int]` then the result type of the target function will be `DList Int`, if the source function's result type is `[a]` then the target function's result type will be `DList a`.

Once the initial state has been created the transformation is performed in a top down, left to right manner with the call to `embRefact` on line 17 of Figure 82. As seen in Figure 85 the definition of `embRefact` simply calls the API function `modMGAltsRHS` which applies a function that modifies an expression to each of the right hand side values of the given function binding. The "real" work of the transformation occurs in `doEmbRefact` which is defined in Figure 86.

The top level, `doEmbRefact`, function controls when the traversal stops, the helper function `doEmbRefact'` is what actually transforms the given expression based on its constructor. The two stop conditions, `b1` and `b2` from Figure 86 lines three and four, are determined by the state of the "type stack".

```
doEmbRefact :: ParsedLExpr -> EmbRefact ParsedLExpr
doEmbRefact expr = do
  b1 <- embDone
  b2 <- skipCurrent
  if b1 || b2
    then do
    lift $ logm "Skipping this expr: "
    lift $ logm (SYB.showData SYB.Parser 3 expr)
    return expr
    else doEmbRefact' expr
  where doEmbRefact' :: ParsedLExpr -> EmbRefact ParsedLExpr
     doEmbRefact' = ...
```

Figure 86: `doEmbRefact` is the function that transforms a given expression to use the target type.

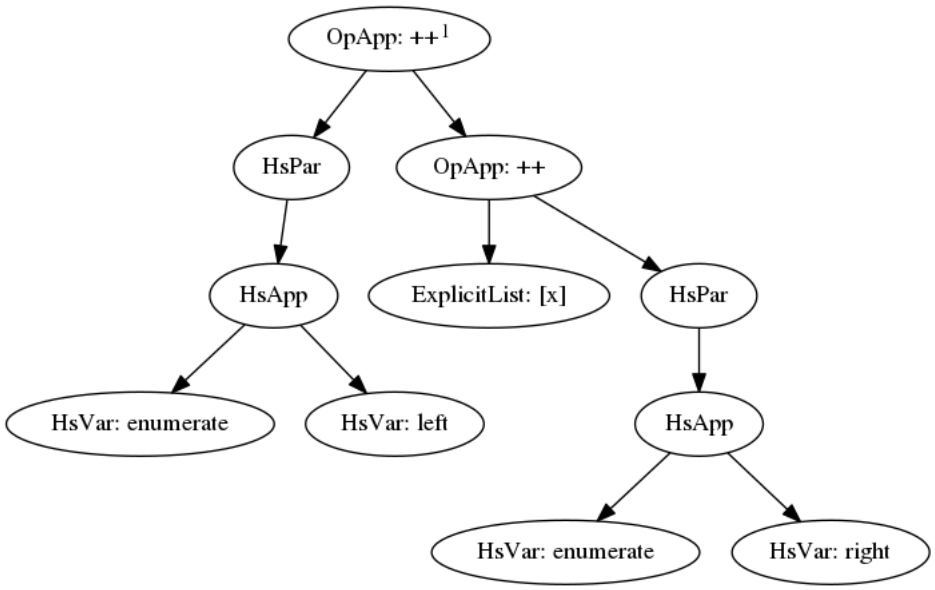
```
data Tree a = Leaf
          | Node (Tree a) a (Tree a)

enumerate :: Tree a -> [a]
enumerate Leaf = []
enumerate (Node left x right) = (enumerate left) ++ [x] ++ (
    enumerate right)
```

Figure 87: Definition of enumerate

5.2.1 The goal type stack

The type stack is part of the refactoring's state and is of type `[Maybe GHC.Type]`. Each element on the stack represents the type that a sub tree must be changed to. The stack will be initialised with a single value "`Just (DList a)`" where `a` is the type of list from the source function this is because the result type of the entire syntax tree needs to be changed to `Just (DList a)`. The stack keeps track the "goal types" of each sub-tree.

Consider the `enumerate` example from section 4.5.5, which is shown here again in Figure 87.

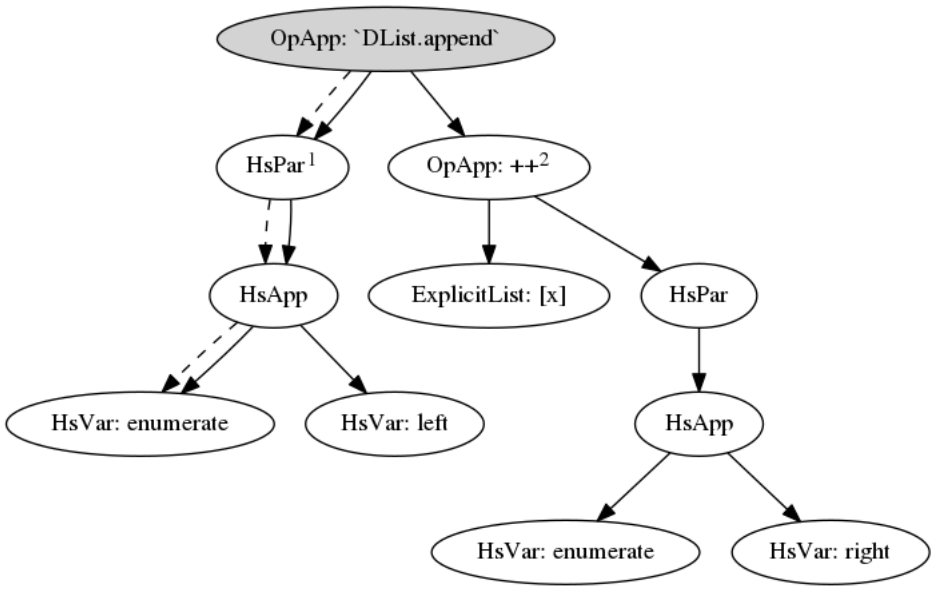



```
1     typeStack = [Just (DList a)1]
```

Figure 88: The initial state of the AST and goal type stack

When refactoring the result type of `enumerate` to become `DList a` the refactoring begins at the root of `enumerate`'s abstract syntax tree and the goal type stack is initialised as `[Just (DList a)]`. At the start of this refactoring the goal type stack holds the single type that the root node needs to become. The initial state of the stack and the abstract syntax tree is shown in Figure 88.

In this case the root node of the AST can be replaced with a call to `DList.append` because this functions result type matches the goal type for the node and `DList.append` is paired with `++` the current value of the node. Once the root node is replaced with the call to `DList.append` the current value of the stack is popped off and `Just (DList a)` is pushed onto the stack twice. This is because `(++)` takes in two arguments both of type `[a]` whereas `DList.append`'s two arguments need to be of type `DList a`. The refactoring continues by changing the root node's left child first. The goal type of this subtree is `DList a` and so the refactoring traverses down to the leftmost child of this subtree. The state of the refactoring at this point in shown in



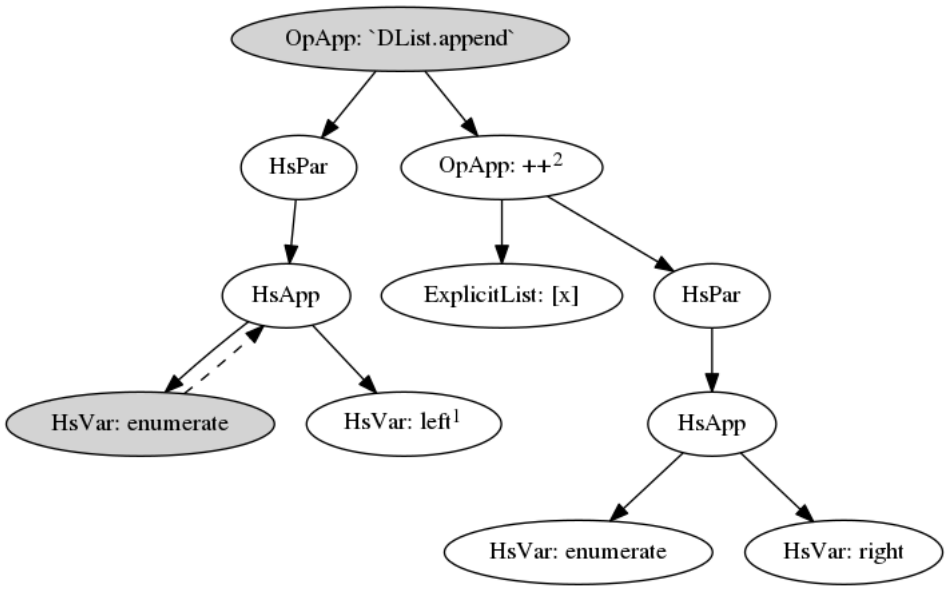

```
1     typeStack = [Just (DList a)[1], Just (DList a)[2]]
```

Figure 89: After replacing the root node the refactoring must resolve the types of its subtrees.

Figure 89.

When the refactoring reaches the call to `enumerate`, it realises that this is a recursive call so the result type is already `DList a` so no change needs to be made and the top of the goal type stack can be popped off. Also since `enumerate`'s argument is not a target of the refactoring, those subtrees will not need to be traversed. This is indicated by pushing `Nothing` onto the stack (see Figure 90). This means that the entire left child of the root node is correct after the change to the root node and the refactoring can begin working on the right child of the root node.

The first node that the refactoring encounters when traversing the right child of the root node is another application of ++. This is handled in the same that its parent node was, with `DList.append` replacing ++ and the new types of its subtrees (`DList a` in this case) pushed onto the stack. In the same way that the root node was handled, the refactoring begins by traversing the left subtree. The state of the refactoring at this



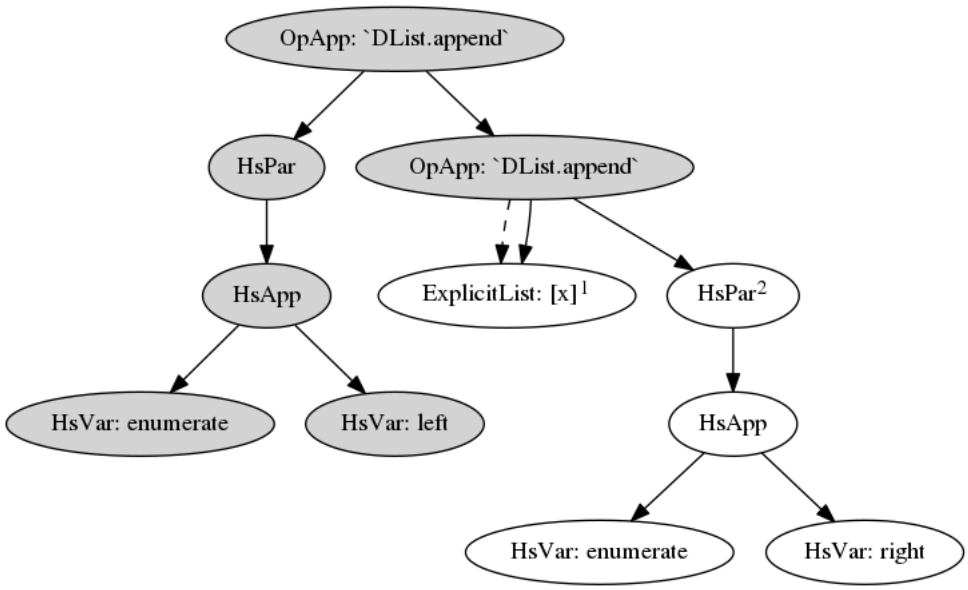

```
1     typeStack = [Nothing1, Just (DList a)2]
```

Figure 90: The `Nothing` on top of the type stack means that the right hand side of the application of `enumerate` doesn't need to be checked. The `Nothing` can be popped off the type stack and the traversal continues upwards.

point is shown in Figure 91.

The node that the refactoring is at is a singleton list expression and according to the goal type stack it must be re-typed to `DList a`. Fortunately the `DList` library provides a `singleton` function that produces a Hughes list of length one from a single parameter. The explicit list can be replaced with a call to `DList.singleton` and this subtree is now correctly typed and the top of the goal type stack can be popped (see Figure 92).

The only part of the AST that hasn't been refactored is the right most expression "`(enumerate right)`". This expression will be handled in the same way that the first recursive call to `enumerate` was. The refactoring will descend to the call to `enumerate`, pop the top of the goal type stack and push `Nothing` onto the stack because none of the arguments need to change. The refactoring will ascend to the `HsApp` node and pop off the `Nothing` value, and therefore not descend to check the




```
1     typeStack = [Just (DList a)[1], Just (DList a)[2]]
```

Figure 91: Traversing the left child again to fix the calls to the newly inserted `DList.append`.

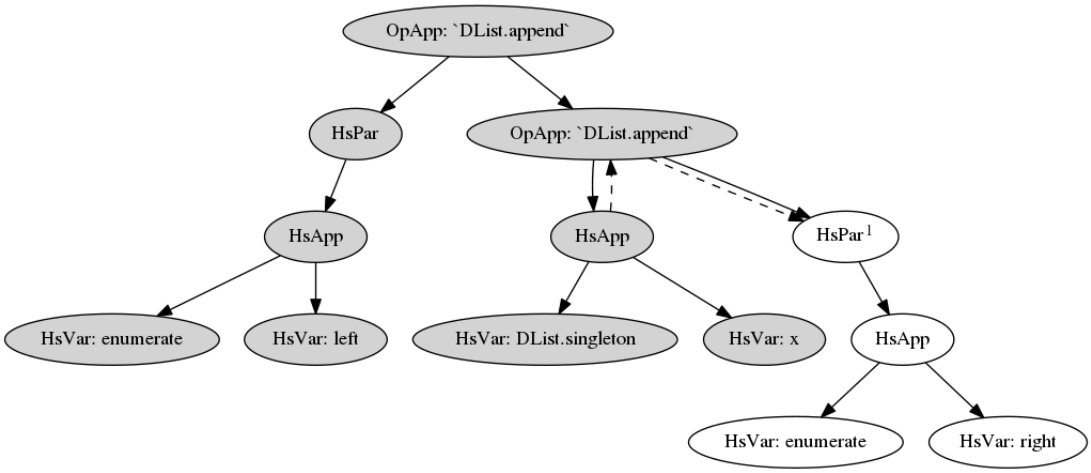
subtree constisting only of the `right` variable. At this point the goal type stack is empty so the refactoring is finished and can return the modified AST.

This section has given a concrete example of how an abstract syntax tree is traversed and how the goal type stack is used. The next section will go through the code that handles each of the different types of expressions.

5.2.2 `doEmbRefact'`: Traversing an expression

Though the top level `doEmbRefact` function determines if a subtree should be changed at all, `doEmbRefact'` determines how the transformation traverses the subtree and the changes that occur based on the constructor of the current abstract syntax node. This section discusses each of the cases of `doEmbRefact'`.

The way that function applications are transformed is shown in Figure 93. The refactoring will transform the left hand side of the application first, then it checks if the




```
1       typeStack = [Just (DList a)[1]]
```

Figure 92: The subtree representing the explicit list has been replaced with a call to `DList.singleton`.

entire application needs to be wrapped with a call to the projection function because the leftmost child of the subtree does not have an equivalent function over the target type. After that the right sub tree is refactored and this refactored expression is applied to the refactored left hand expression.

If the projection function needs to be introduced at this layer then another application is wrapped around the already modified syntax tree. Otherwise the application with its modified sub trees can just be returned.

The next case, operator application, is handled much like function application is: whether or not the operator can be replaced by a target type equivalent determines if the projection function needs to be added. Otherwise the left- and right-hand expressions can be refactored.

The new operator may not be allowed in an infix position and in this case backquotes (`) are added, which in Haskell makes a prefix function into an infix version of that function. A good example of this is `DList.append`. `DList.append` is paired with `(++)` which is an operator whereas `DList.append` is a prefix function, therefore the refactoring of "`leftList ++ rightList`" is "`leftListRef`

```
1       doEmbRefact' (GHC.L l (GHC.HsApp le re)) = do
2         le' <- doEmbRefact le
3         wrapWithProj <- shouldInsertProj
4         re' <- doEmbRefact re
5         lift $ logm "POST RHS REFACT IN APP CASE"
6         let newApp = (GHC.L l (GHC.HsApp le' re'))
7         if wrapWithProj
8           then do
9           projRdr <- getProjFun
10          let var = GHC.HsVar projRdr
11          pApp <- lift $ wrapInPars newApp
12          lVar <- lift $ locate var
13          lift $ addAnnVal lVar
14          let fullApp = GHC.HsApp lVar pApp
15          lift $ locate fullApp
16          else return newApp
```

Figure 93: The case of `doEmbRefact'` that handles function applications.

```
1       doEmbRefact' (GHC.L l (GHC.OpApp le op rn re)) = do
2         op' <- doEmbRefact op
3         wrapWithProj <- shouldInsertProj
4         lift $ addBackquotes op'
5         le' <- doEmbRefact le
6         re' <- doEmbRefact re
7         let newOp = (GHC.L l (GHC.OpApp le' op' rn re'))
8         wrapProjIfNeeded wrapWithProj newOp
```

Figure 94: The case of `doEmbRefact'` that handles operator applications

`` `DList.append` rightListRef`` where `leftListRef` and `rightListRef` are the refactored parameters to the append function.

The major changes that the refactoring makes are performed when `doEmbRefact'` encounters a variable. This logic is shown in Figure 95. The first six lines of this case (lines 2 through 7 in Figure 95) retrieve the type of the current variable from the typed abstract syntax.

Next (line 10 of Figure 95) the refactoring checks if there is a possible replacement

```
1       doEmbRefact' var@(GHC.L l (GHC.HsVar rdr)) = do
2         st <- get
3         let ts = typeStack st
4             fs = funcs st
5         typed <- lift getRefactTyped
6         mId <- lift (getIdFromVar var)
7         let id = gfromJust ("Tried to get id for: " ++ SYB.
              showData SYB.Parser 3 rdr) mId
8             currTy = GHC.idType id
9             keyOcc = GHC.rdrNameOcc rdr
10            mVal = (GHC.occNameString keyOcc) `M.lookup` (eqFuns
                  fs)
11        case mVal of
12          Nothing -> do
13            popTS
14            dontSearchSubTrees currTy
15            lift $ logm "Nothing case of mVal"
16            printStack
17            insertProjToT
18            return var
19          (Just (oNm, ty)) -> do
20            let changedTypes = typeDifference ty currTy
21                newE = (GHC.L l (GHC.HsVar oNm))
22            oldAnns <- lift fetchAnnsFinal
23            case M.lookup (mkAnnKey var) oldAnns of
24              Nothing -> lift (mergeRefactAnns $ copyAnn var
                    newE oldAnns)
25              Just v -> do
26                let dp = annEntryDelta v
27                lift $ addAnnValWithDP newE dp
28            popTS
29            addToTS changedTypes
30            return newE
```

Figure 95: The case of `doEmbRefact'` that handles variables

function. If a replacement is not available then the transformation can pop the top element from the stack and any parameters that are applied to this value don't have to be searched. This is done through the call to `dontSearchSubTrees` which puts `Nothing` values onto the type stack. For example, if the current variable was "`f :: a -> b -> c`" and there was no possible replacement for `f`, then

`dontSearchSubTrees` would push two `Nothing`s onto the type stack so that the subtrees of type `a` and `b` aren't modified. Finally in this case the boolean "`insertProj`" in the refactoring state is flipped, which informs the refactoring that higher up the tree the application of this value needs to be wrapped with the projection function.

When a possible replacement for the current value is found (this case begins on line 19 of Figure 95), the refactoring first calculates which parameters of the new function have different types from the original function which is stored in the `changedTypes` variable. This variable is a list of `Maybe GHC.Type`s where the `Nothing` values mean that the type of that parameter has not changed and "`Just _`" values indicate the new type that the parameter must become. This list represents each parameter of the variable from left to right order. This list of type changes that need to be handled is pushed onto the type stack in line 29 of Figure 95 after the current goal type is popped from the stack.

The rest of this case constructs the new expression and copies over the annotation from the old piece of abstract syntax so that any formatting associated with the original call is preserved in the new source code. Finally the new expression can be returned.

One of the goals that influenced the design of this system was to produce a reusable system so that more embeddable type refactorings could easily be created. A tricky situation arises with Haskell's support for explicit list syntax (e.g. "`[1,2,3]`"). Because lists are the source type of the "list to Hughes list" refactoring, the explicit list syntax may need to be transformed during the refactoring, so this type-specific bit of code remains in the otherwise generic function. Ideally this function could remain completely type agnostic but because the "list to hughes list" refactoring works over the list type it needs to descend into this type specific constructor of the abstract syntax tree.

The case that handles explicit lists is shown in Figure 96 and is simple enough. If the explicit list is just a single element long (e.g. "`[3]`") then that value will be passed to the difference list's `singleton` function, otherwise the projection function is wrapped around longer lists.

```
1       doEmbRefact' eLst@(GHC.L l (GHC.ExplicitList ty mSyn lst
           )) = do
2         if (length lst) == 1
3           then do
4           st <- get
5           let fs = funcs st
6              singletonRdr = mkQualifiedRdrName (GHC.
                  mkModuleName "DList") "singleton"
7              singletonVar = (GHC.HsVar singletonRdr)
8           lVar <- lift $ locate singletonVar
9           lift $ addAnnVal lVar
10          lift $ zeroDP lVar
11          let rhs = head lst
12          lift $ setDP (DP (0,1)) rhs
13          lApp <- lift $ locate (GHC.HsApp lVar rhs)
14          parApp <- lift $ wrapInPars lApp
15          return parApp
16          else do
17          st <- get
18          let fs = funcs st
19             projRdr = projFun fs
20          lVar <- lift $ locate (GHC.HsVar projRdr)
21          lApp <- lift $ locate (GHC.HsApp lVar eLst)
22          lift $ wrapInPars lApp
23          return lApp
```

Figure 96: The case that handles explicit list syntax

```
1 doEmbRefact' ex = gmapM (SYB.mkM doEmbRefact) ex
```

Figure 97: The catch all case of `doEmbRefact'`

Finally `doEmbRefact'` has a simple catch-all case that handles all the other constructors of `HsExpr`. This case shown in Figure 97 uses the `gmapM` generic combinator to recursively call `doEmbRefact` on all of the given expression's children.

5.3 Other enhancements made to HaRe

Another contribution made in the process of creating the refactorings described in this thesis was many enhancements to HaRe's API. With the switch to the GHC API and the introduction of ghc-exactprint, a large part of HaRe's original API was reimplemented but the functionality it provided was designed for aiding the development of refactorings that used Programatica's abstract syntax tree instead of GHC's. Additionally ghc-exactprint has a very different way of formatting source code and the current API did not help refactoring developers use this library. This section will describe contributions I made to the HaRe's API that help working with both GHC's abstract syntax and ghc-exactprint.

These contributions took two forms "queries" and "transformations." Queries extract parts or determine properties of an AST, whereas transformations are small modifications to source code that are useful to many refactorings. This section will briefly describe some of the functionality provided by HaRe's API that I implemented while developing the refactorings described in this thesis.

5.3.1 The `Query` module

One thing that many refactorings have in common is the need to extract the same AST elements. In an effort to reduce the amount of duplicated code, the `Query` module was created to hold functions that extract or check properties of syntax elements.

Common queries do things like retrieve the function binding based on function name, as seen in Figure 98. Many refactorings have function names as a input argument so `getHsBind` is useful for retrieving the appropriate binding.

Currently the `Query` module defines the following functions:

- `getVarAndRHS` retrieves the pattern and right-hand side expression from a function match
- `getHsBind`

```
1  getHsBind :: (Data a) => GHC.RdrName -> a -> Maybe (GHC.HsBind
       GHC.RdrName)
2  getHsBind nm a = SYB.something (Nothing `SYB.mkQ` isBind) a
3     where
4  #if __GLASGOW_HASKELL__ <= 710
5        isBind ((bnd@(GHC.FunBind (GHC.L _ name) _ _ _ _ _)) ::
             GHC.HsBind GHC.RdrName)
6  #else
7        isBind ((bnd@(GHC.FunBind (GHC.L _ name) _ _ _ _)) ::
             GHC.HsBind GHC.RdrName)
8  #endif
9            | name == nm = (Just bnd)
10       isBind _ = Nothing
```

Figure 98: This function retrieves the function binding of the given name from the provided syntax tree.

- `getFunName` takes in a string representation of a function name and gets the `Name` of that function
- `getTypedHsBind` given an `OccName` retrieves the typed syntax tree for a function binding
- `getTypeSig` gets the type signature of a function based on the location of the function binding and its name
- `isHsVar` given a string and an expression this function checks if the expression is a variable with the same name as the value of the string
- `astCompare` does a rough estimate of whether two abstract syntax trees are the same. This is done by checking that the trees are the same shape and have the same constructors, and that any names that appear in the trees are the same
- `lookupByLoc` retrieves the syntax element at a given location
- `getIdFromVar` takes in a parsed expression that uses the `HsVar` constructor and retrieves the typed `Id` of that variable

5.3.2 The `Transform` module

The other major contribution made to HaRe's API is the addition of the `Transform` module. This module collects small changes to the syntax tree and annotations that are not specific to any particular refactoring and need to be performed by multiple refactorings.

Abstract syntax changes like adding a import declaration to the target module, or replacing the original binding of a function with another one are small changes to source programs that many refactorings need to do. The other type of transformations that refactorings all perform do not affect the abstract syntax tree but change the formatting of the program. In HaRe this formatting work is done by modifying the annotations associated with the syntax elements.

The `Transform` module defines the following functions:

- `addSimpleImportDecl` adds a new import to the current module based on a string module name and an optional qualifier.
- `wrapInLambda` creates a lambda expression from a pattern and right-hand side expression and also creates all the necessary annotations so that the expression displays correctly.
- `wrapInParsWithDPs` wraps the given expression in parentheses and offsets the new expression based on the given "DPs" (delta position).
- `wrapInPars` is similar to `wrapInParsWithDPs` but defaults to a single column offset between the new expression and the syntax element that comes before it.
- `removePars` removes parentheses from an expression and makes sure that the new expression is offset one column from the previous syntax element.
- `addNewLines` adds newlines before the given syntax element.

- `replaceTypeSig` replaces the existing type signature of a function with the given one.
- `replaceFunBind` replaces the source binding of a function with the given one.
- `addBackquotes` adds backquote characters around an expression.
- `constructHsVar` constructs an `HsVar` expression from a name.
- `constructLHsTy` constructs a type variable from a name.
- `insertNewDecl` inserts a new declaration into the current module from a given string that represents that declaration.
- `rmFun` removes a function from the current module based on a name.

5.4 Summary

This chapter has described how the generalising `Maybe` and the list to Hughes list refactorings have been implemented in HaRe. In addition to the specific implementations of these two refactorings this chapter also described the additions to HaRe's API that were made to support the refactorings developed for this thesis. In addition to the enhancements made to HaRe's general API for refactoring, this chapter also covers the API that supports reversibly embeddable type refactorings such as the list to Hughes list refactoring.

Now that the the basic design and implementation of data-driven refactorings have been introduced the thesis will proceed by looking at two more refactorings, generalising monadic code to applicative functors (Chapter 6) and the monadification of pure code (Chapter 7).

Chapter 6

Generalising Monadic Code to Applicative Functors

The previous chapter introduced the concept of a functional data refactoring and gave three examples of data driven refactorings: introducing a type synonym, generalising Maybe, and list to Hughes lists. This chapter will cover another generalising refactoring, generalising monadic functions to use applicative functors if certain conditions are met.

6.1 Applicative Functors

In their 2008 functional pearl "Applicative Programming with Effects, (McBride and Paterson 2008)" Conor McBride and Ross Paterson introduced a new type class that they called Idioms but are more commonly known as Applicative Functors. Applicative functors, like monads, provide a way to run computations within some context, and collect their results. Applicative functors are more expressive than functors but more general than monads.

Applicative functors are implemented in GHC as the type class `Applicative`.

Interestingly, even though McBride and Paterson proved that all monads are also applicative functors in their original functional pearl (McBride and Paterson 2008), 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 is now a large body of code which could potentially be rewritten using the applicative operators rather than the monadic ones.

This chapter will discuss the design and implementation of a refactoring which automatically refactors code written in a monadic style to use the applicative operators instead given it meets certain conditions that we will spell out. Section 6.2 is a brief overview of the operations of the `Applicative` typeclass, Section 6.3 discusses the applicative programming style and, in general, how programs are constructed using the applicative operators. Next, Section 6.4 describes how the Haskell community uses the applicative typeclass, and Section 6.5 describes in detail the refactoring and its preconditions. Section 6.6 discusses the refactoring's implementation in HaRe and some additional refactorings that can be used to transform the code to meet the preconditions. Finally some applications that are good candidates for this refactoring are discussed in Section 6.7.

6.2 The Applicative Type Class

This section is a short introduction to the `Applicative` type class and some related functions that are commonly used to write functions in an applicative style. To discuss applicative functors we first have to cover functors. Functors originate from category theory and are a natural generalisation of maps over lists. The definition of GHC's `Functor` type class is shown in Figure 99. The `Functor` type class defines a single function that must be implemented: `fmap`.

`fmap` allows for a pure function to be applied to the values inside the functor `f`. Essentially `fmap` as a function that runs other functions within the `Functor`'s context.

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

Figure 99: The `Functor` type class

```
1 g :: a -> b -> c
2 x :: f a
3 y :: f b
```

Figure 100: A function and two functors.

Functors do have a serious limitation, consider Figure 100 which shows the types of the function `g` and two values `x` and `y`, which are some functor `f` that contains values of type `a` and `b` respectively. With these three values the obvious goal would be to compute a value of type `f c`. When `g` is mapped over `x`, a value of type `f (b -> c)` is returned.

Unfortunately there is no way to either extract the function of type `(b -> c)` from the fuctor or apply `f (b -> c)` to the values in `y` using the `Functor` type class alone. Sequencing commands in this way requires a more powerful abstraction. Applicative functors allow for this sort of sequencing of commands within a context (O'Sullivan, Goerzen and Stewart 2008).

In Haskell applicative functors are implemented by the `Applicative` type class. `Applicative` declares two functions, `pure` and `(<*>)`, the types of these two functions are shown in Figure 101.

The `pure` function is the same type as monad's `return`: it simply lifts a value into the applicative context. The other function `(<*>)`, which is typically pronounced "applied over" or just "apply", takes in two arguments, both of which are applicative values[1]. The first argument is a function, from types `a` to `b`, within the applicative context, and the second argument is an `Applicative` over type `a`. Apply will return an `Applicative` over `b`.

[1]That is these values are of a type that is an instance of applicative.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Figure 101: `Applicative`'s minimal complete definition

```
(*>) :: Applicative f => f a -> f b -> f b
a1 *> a2 = (id <$ a1) <*> a2

(<*) :: Applicative f => f a -> f b -> f a
(<*) = liftA2 const

(<$) :: Functor f => a -> f b -> f a
(<$) = fmap . const

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)
```

Figure 102: Variations on apply.

6.2.1 Other useful functions

Though `pure` and apply are the only 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 type classes which will be briefly covered here. First, there are two special cases of apply, whose definitions are shown in Figure 102 along with the definitions of some helper functions.[2]

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 an operation affects the applicative context but their result will not affect the overall result of the applicative expression.

The function `f`, from Figure 103, looks up a customer data type from the `IO` context and logs all of the customer ids that are looked up. We don't really care what

[2] `liftA2` and `(<$)` are used in these definitions for performance reasons. The reasoning behind these decisions is described in the comments in the declaration of the `Applicative` type class, which can be found here: `http://hackage.haskell.org/package/base-4.11.1.0/docs/src/GHC.Base.html#Applicative`

```
f :: Int -> IO Customer
f id = writeToLog id *> getCustomerById id
```

Figure 103: A function that logs its argument before returning the result.

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

Figure 104: All `Applicatives` have this property.

`writeToLog` returns but we do want the effects that it causes (writing a log file) so we compose the calls to `writeToLog` and `getCustomerById` with the `*>` operator to cause both of the functions' effects to happen but only return the value of `getCustomerById`.

Another useful function is the infix version of `fmap`, `(<$>)`. A consequence of the applicative laws is that the functor instance of every `Applicative` will satisfy the equation in Figure 104 (McBride and Paterson 2016). This means that instead of explicitly lifting a pure function, the infix version of `fmap` can be used instead. The next section will cover how these functions can be used in an applicative style of programming.

6.3 The Applicative Programming Style

McBride and Paterson proved that any expression built from the applicative combinators can take the canonical form in Figure 105 (McBride and Paterson 2008):[3] The `is`'s in Figure 105 are computations within the applicative context. These computations are the arguments of some pure function "`f`" that has been lifted into the context with the `pure` function.

Context-free parsing is a good "use case" of the applicative style and many examples in this chapter are taken from parsers defined using the parsec library (Leijen and

[3]This notation is taken from (McBride and Paterson 2008) where applicatives were called "idioms", hence the use of "`is`".

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

Figure 105: The applicative functor in canonical form.

```
data Currency = Dollar
              | Pound
              | Euro

data Money = M Currency Integer Integer

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

Figure 106: Parsing Money

Martini 2006). The first example of the applicative programming style is a function, from Figure 106, that parses money amounts of the form:

`<currency symbol><whole currency amount>.<decimal amount>`

This parser would recognise "$4.59" or "£64.56" and parse it into the `Money` data type as defined in Figure 106. The `parseMoney` function is in the canonical form as defined by McBride and Paterson. 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 respectively.

The only difference between the `readWhole` and `readDecimal` is that `readDecimal` has to consume the decimal point before reading the number. Instead of duplicating that number code, a small refactoring that lifts the parsing of the decimal into the `parseMoney` function will allow us to reuse the `readWhole` function, as seen in Figure 107. In this definition of `parseMoney` the parsed decimal character is discarded because of the use of `<*` rather than the normal apply.

All of the variations of apply[4] are left associative so that the definition of `parseMoney` in Figure 108 causes a type error. This error can be corrected by parenthesising the subexpression:

[4] As in `(<*>)`, `(<*)`, and `(*>)`.

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

Figure 107: An alternate `parseMoney` definition.

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

Figure 108: A non-well typed definition of `parseMoney`.

```
char '.' *> readWhole
```

The canonical style of applicative functions, as defined by (McBride and Paterson 2008), may not be the most idiomatic way to define things. The function from Figure 109 parses strings surrounded by double quotes. `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 prepending "`id <$>`", this does not change the behaviour of the function but it shows that any expression constructed using the applicative operators can be transformed to the canonical form.

The examples covered in this section give a basic introduction to programming in an applicative style. Next we will discuss how the Haskell community has begun to use `Applicatives` in practice.

6.4 Applicative in practice

It is helpful to understand how the Haskell community has adopted and uses the applicative interface to help inform the development of the refactorings in this chapter. If the Haskell community has fully embraced the addition of applicative functors and use them regularly, then a refactoring that rewrites the monadic operations to use applicative ones is not as useful.[5]

[5]It could still be useful for updating a legacy code base.

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

Figure 109: A parser for string literals.

```
pure = return
(<*>) = ap
```

Figure 110: The suggested implementation of `Applicative`

The simplest question we wanted to answer was how many projects are actually taking advantage of the `Applicative` interface versus using the suggested implementation. When `Applicative` was made a superclass of `Monad` an implementation of the `Applicative` interface was suggested (Luposchainsky 2014). This implementation allowed programmers to get their existing `Monad` instances to re-compile successfully with minimum effort. We are interested in how many instances use this suggested implementation, which is defined in Figure 110.

6.4.1 Methodology

Hackage is the main package repository for open source Haskell code (GHC 2016). The code in Hackage is broken up into 12,601 packages[6] and within these packages there are 144,413 Haskell source files, of which 133,246 (about 92% of the total) were successfully parsed and searched.

Parsing failures were caused by two main reasons. First *ghc-exactprint* cannot parse C or C++ code so any files with embedded C/C++ code blocks were not searched. The second major reason for failure was modules not including language pragmas at the top of each file. A feature of the Cabal build system allows turning on language extensions at the package level rather than the module level: since we are parsing individual files rather than entire projects the parser does not realise that certain language extensions may be turned on. Parsing failures only happened about in about 8% of the files so

[6] As of 14 May 2018.

there was no great need to modify ghc-exactprint to handle these cases.

Within each package every "`.hs`" file was parsed using the *ghc-exactprint* library.[7] Once we had the abstract syntax tree we could use generic traversal combinators provided by *Scrap Your Boilerplate* (SYB) to search the parsed files for implementations of the `Monad` and `Applicative` type classes. From the successfully parsed files a total of 2,376 `Applicative` and 2,126 `Monad` instance declarations were found.

6.4.2 Search Results

Once all the `Applicative` instances were discovered it was much easier to search through them to try and understand how the Haskell community has been using this type class. The `Applicative` type class is an interesting case of programming language design and management. A language feature, in this case `Monads`, was accepted and even celebrated by the language's community. Then, twenty-five years into the language's life, to continue use this popular feature users have to modify their code.

The question that this search is trying to answer is: how has the Haskell community adapted to having to write `Applicative` instances for every `Monad` instance? Are programmers taking advantage of what the new type class can offer?

So, how has the Haskell community adapted to the forced introduction to applicative functors? The reaction seems to be mixed. Of those 2,376 `Applicative` instances the search discovered 841 directly defined apply as: `(<*>) = ap`. This is the suggested implementation for apply.

Beyond the instances of `Applicative` that simply define apply to be `ap` there are several other instances that essentially copy the definition of `ap`. For example, Figure 111 shows the `Applicative` definition of `Sink` from the Sousit library. This definition is an exact duplicate of the definition of `ap`. There are other ways that this functionality can be duplicated, for example, the applicative instance in Figure 112 shows another pattern that several packages used.

[7] Unfortunately *ghc-exactprint* cannot parse literate Haskell ("`.lhs`") files so they were excluded

```
instance Monad m => Applicative (Sink i m) where
   pure = return
   af <*> s = do
      f <- af
      v <- s
      return (f v)
```

Figure 111: The applicative instance of `Sink` from *Sousit* version 0.4.

```
instance Applicative TypeCheckMonad where
   pure x  = TCM $ \_ _ _ -> Right x
   mf <*> mx = mf >>= \f -> fmap f mx
```

Figure 112: The applicative instance of `TypeCheckMonad` from *Hakaru* version 0.6.0.

There could be other `Monad` and `Applicative` instances that this search couldn't pick up. With the `GeneralizedNewtypeDeriving` language extension `Applicative` and `Monad` instances can be automatically derived by the compiler with their most obvious implementations (Bajt 2014). Though it is unclear how often this extension is used, as there seems to be some community pushback about using it because it could cause unsafe type coercions (Philip 2012).

One interesting thing that the search discovered is that only 697 instances of `Applicative` defined `pure` as `return`. This is fewer than the number of instances that define apply as `ap`. It is surprising that implementers take the time to define a pure instance but then define apply as `ap`. Though it's impossible to know exactly why people are doing this, it maybe has to do with the Haskell community's understanding and comfort with how to define `pure` for their types versus apply.

In total this search discovered 868 instances of `Applicative` where either apply was directly defined to be `ap` or `ap`'s definition was replicated in some way. This means that roughly 35% of the `Applicative` definitions simply use the suggested implementation. The following subsection will look at some other common patterns that go beyond the most basic `Applicative` implementation.

```
instance Applicative Pair where
 pure a = Pair (a, a)
 (<*>) (Pair (fa, fb)) (Pair (a, b)) = Pair (fa a, fb b)
```

Figure 113: The applicative instance from *SGplus* version 1.1

```
instance Applicative m => Applicative (DockerT m) where
   pure a = DockerT $ pure a
   (<*>) (DockerT f) (DockerT v) = DockerT $ f <*> v
```

Figure 114: Applicative instance from *Docker* version 0.3.0.0

6.4.3 Common Implementation Patterns

In addition to searching how many instances use the standard implementation, the search also was used to discover the different ways `Applicative` instances were being implemented. Are there any general patterns that emerge?

Finding general patterns like this is a much more complex task than just searching for calls to `ap` in the definition of apply. All of the Applicative implementations were written into a report and the report was manually analysed for implementation patterns that cropped up multiple times.

The first common implementation pattern simply pattern matches on the type's constructor to unwrap inner values and apply them to each other. The `Applicative` instance definition in Figure 113 is an example of this approach.

Another pattern that was used several times constrains an inner type to be `Applicative` as well and just passes off the work to that inner `Applicative` instance. The example in Figure 114 shows how this can be defined.

6.4.4 Conclusions

The search results are inconclusive. It is hard to make firm judgements about people's familiarity with a complex type class just from the implementation of the type class

```
parseStr :: CharParser () String
parseStr = do
  char '"'
  str <- many1 (noneOf "\"")
  char '"'
  return str
```

Figure 115: A string literal parser.

alone. However taken in combination with other results, such as the work done in Marlow et al. (2016), which takes the approach that a single universal notation is easier to use, and the long history of people having trouble understanding monads (Yorgey 2009a), it appears that programmers do struggle with type classes like monads and applicative functors. Additionally, the popularity of the default `Applicative` definition could suggest that the Haskell community has not completely adjusted to the type class.

At the same time it is possible that most monad instances are more monadic than applicative and so spending time defining unique applicative instances may only be useful for a few types. The rest of this chapter will describe this refactoring, its implementation, preconditions, and some related refactorings that transforms code so that it may be able to pass the preconditions. The next section will describe the design of this refactoring.

6.5 Refactoring Monadic Programs to use Applicative

This section will describe how the refactoring of monadic code to the applicative style works via a series of examples. Many of these examples are taken from case studies of the parser for money amounts and a JSON parser.[8]

[8]The full source code for these parsers can be found at `https://goo.gl/mrCdFh` and `https://goo.gl/ysWmSS`

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

Figure 116: The refactored string literal parser

The first example will refactor a parser for string literals, which is defined in Figure 115. This parser first consumes a double quote character(`char '"'`) then parses one or more characters that are not double quotes and assigns those characters to the variable named `str`[9]. Finally the closing quote is consumed and `str` is returned. The refactored version of the function is shown in Figure 116.

The refactoring in this case is relatively simple. Since the third and fifth lines of the original function are not bound to a variable, we know that the values returned by those functions will be ignored, hence the use of the `*>` and `<*` operators. This way the double quote characters will still be parsed but the values returned by those parsers will be discarded. The original function returns the variable `str`, without calling any pure functions on it. There will therefore be no pure function application to the left of the effectful functions.

This is a straightforward function to convert to applicative style. The next example is a little more complex. The function in Figure 117 is a slight modification of the money parser from Section 6.3 as well. This version of the parser makes the decimal amount optional, so it will recognize both "£10.35" and "€4".

The `parseMoney` function parses text into the `Money` data type from Section 6.3. The `option` parser allows for the decimial to become optional. The `option` combinator will first attempt to apply the parser which is given as its second argument. If this parser fails then `option` returns the value it was given in its first argument.

The return statement in the monadic definition of `parseMoney` calls the constructor `M` with the three values that come from the monadic context. Since `M` is a pure

[9]This line is composed of two parser combinators, `many1`, and `noneOf`. `many1` takes another 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; that character is then returned.

```
1
2 parseMoney :: CharParser () Money
3 parseMoney = do
4   currency <- parseCurrency
5   whole <- many1 digit
6   decimal <- (option "0" (do {
7                     char '.';
8                     d <- many1 digit;
9                     return d}))
10  return $ M currency (read whole) (read decimal)
```

Figure 117: `parseMoney` version 2.

```
1 parseMoney :: CharParser () Money
2 parseMoney = M <$> parseCurrency
3              <*> (read <$> many1 digit)
4              <*> (read <$> option "0" (do {
5                     char '.';
6                     d <- many1 digit;
7                     return d}))
```

Figure 118: The first attempt at refactoring `parseMoney`.

function, it is placed at the front of the chain of applicative operations, composed with `(<$>)`. The `whole` and `decimal` variables are passed as arguments to `read`, a pure function, so these values will be composed with `read` using the `(<$>)` operator. The refactored definition of `parseMoney` is defined in Figure 118.

The inner `do` block that is passed as the second argument to `option` is also a valid target of this refactoring. The result from applying this refactoring to this `do` block is shown in Figure 119.

6.5.1 Splitting the Applicative Chain

As chains of applicative computations can get more complicated it is useful or indeed mandatory to parenthesize parts of the function. The `objEntry` function in Figure 140 parses a single entry from a JSON object. Each entry consists of a string key and a value

```
parseMoney :: CharParser () Money
parseMoney =
  M <$> parseCurrency
    <*> (read <$> many1 digit)
    <*> (read <$> option "0" (char '.' *> many1 digit))
```

Figure 119: `parseMoney` after inner do refactoring.

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

Figure 120: A JSON object parser.

which can be any valid JSON value, separated by a colon, and stores the key and value in a tuple.

When the applicative chain has a number of functions composed with the `(*>)` or `(<*)` operators, there are often multiple valid ways to add parentheses to the function. For example, the two definitions of `objEntry` in Figure 121 are equivalent.

The refactoring will produce the first version of this function, and in general will group each of the value-producing expressions (`parseStr` and `parseJVal` in this case) with the closest non-value producing statements. Every group contains a single value producing expression and zero or more non-value producing expressions on either side of the value producing expression. Each of these groups will then be surrounded by parentheses if it contains non-value producing statements, and the groups will be composed using the `<*>` operator. A more formal description of this is provided in Section 6.6.

6.5.2 Preconditions

This refactoring has two (simple) preconditions: first, the target function must be definable with the applicative interface (which will be explained below) and secondly, the order that variables are bound in the `do` block must correspond to the order in which

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

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

Figure 121: Different ways to separate the applicative chain.

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

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

Figure 122: The types of apply and bind

they are used in the return statement.

What does this first condition actually mean? Where is the line between applicative functors and monads? Let's start by comparing the type signatures of the bind and apply functions, they are shown in Figure 122. One thing to keep in mind is that these two functions' arguments are in opposite order where the function type is the first argument to apply whereas the second argument to bind is a function. The difference between bind and apply is in the type of this function argument. The second argument of bind is 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 returned by a computation in the monad, while all of the arguments to apply are fully within the applicative context.

What does this mean in practice? Take for example the function `f` from Figure 123. The function `f` first parses a number `n` then parses `n` "things." This function is not definable using applicative functors because `n` is an argument to `parseNThings` as a pure value. The implementation of the first precondition checks that any values bound

```
f = do
  n <- parseN
  xs <- parseNThings n
  return xs
```

Figure 123: A context dependent parser.

```
g :: Monad m => m (A,B)
g = do
  b <- getB
  a <- getA
  return (a,b)
```

Figure 124: A function with out of order bindings

in the `do` statement are not used in a right hand side expression before the `return` statement.

The second precondition is a bit more intuitive to grasp. Consider the function `g` in Figure 124. The naive implementation of the refactoring is in Figure 125. This version of the refactoring maintains the order of statements from the `do` block but this causes a type error. Instead, if the refactoring swaps the order of the statements (Figure 126) the order that the contextual effects are performed changes, and the meaning of the refactored program is no longer the same as the source program so the refactoring is not valid.

These preconditions enforce subtly different things. The first precondition, that the target function needs to be expressible with the `Applicative` interface, is the obvious requirement that the target function needs to meet for the refactoring to be able to be performed successfully. The entire premise of this refactoring is to convert programs using the monadic interface to instead use the more general applicative one, so the target function must be expressible using that interface.

The second precondition, especially when described as naively as it has been in this section, seems to also be mandatory for the refactoring to not change the behaviour of the target function. However, an additional transformation could be applied to `g`

```
1 g :: Applicative a => a (A,B)
2 g = (,) <*> getB <*> getA
```

Figure 125: This attempt at refactoring causes a type error.

```
1 g :: Applicative a => a (A,B)
2 g = (,) <*> getA <*> getB
```

Figure 126: This refactoring changes the order the effects happen in.

that would allow the generalisation refactoring to continue without type errors and preserve the ordering of effects, the introduction of a lambda expression into the `return` statement. The details of this transformation are described in Section 6.5.3.3.

The next section will discuss further refactorings that help modify potential target functions so that they can meet the preconditions discussed in this chapter. The second precondition can always be "avoided" if the programmer so chooses by applying an additional refactoring that wraps the expression that is applied to `return` in a lambda expression. The first precondition on the other hand cannot be simply avoided. The next section will discuss refactorings that can help maximise the amount of code that can be refactored, but code that does not pass the first precondition cannot be refactored.

6.5.3 Additional Refactorings

The preconditions described in the previous section, and the second one in particular, may seem overly strict and needlessly limit the scope of code to which the refactoring could be applied. Instead of developing a single monolithic refactoring that contains many separate cases, the approach that HaRe takes is to develop multiple small refactorings that can be composed together. Composing refactorings together in this way is known as a composite refactoring (Li and Thompson 2011).

This section will describe a number of refactorings that will help transform (or "prefactor") code so that it is capable of passing the preconditions for the generalising

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

Figure 127: The function `f`, lines three through five can be extracted.

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

Figure 128: `g` was extracted from the original function `f` in Figure 127

.

applicative refactoring.

6.5.3.1 Extract monadic code

The `f` function in Figure 127 will not pass the preconditions because both `b` and `y` are extracted from the monadic context and used in a right hand side expression. However, lines three through five do not really affect the rest of the function so they could be refactored to their own function. The result of extracting these lines into their own function, `g`, is shown in Figure 128. Now `f` is a valid target for generalising to the `Applicative` interface, the final version of this function is in Figure 129.

The extraction of monadic code does not have to create a top level variable; if the developer preferred the extracted definition could be in a `let` or `where` clause instead.

```
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
```

Figure 129: The final result of the refactoring with `f` rewritten using the `Applicative` interface.

```
1 f = do
2   x <- result1
3   y <- result2
4   z <- result3
5   log z
6   return (x,y)
```

Figure 130: Lines four and five can be merged into a single do block.

6.5.3.2 Inline do blocks

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

Consider the function in Figure 130, normally the variable `z` being used again in line five would prevent the function from being refactored. Merging lines four and five into an inline do block would allow this function pass the preconditions as seen in Figure 131. Now the entire function can be refactored to the version shown in Figure 132. Additional refactoring can then take place with the inner `do` block also undergoing refactoring, see Figure 133

6.5.3.3 Introduce lambdas to reorder statements

The other precondition to the "generalise applicative" refactoring is the condition that all bound variables need to be extracted from the monadic context in the order they appear in the return statement. Reconsider the example given at the end of Section 6.5.2

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

Figure 131: `f` with two lines merged into an inline do block.

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

Figure 132: The inline do allows the function to be rewritten with the applicative operators.

(repeated in Figure 134).

Since `a` appears before `b` in the return statement but `b` is bound before `a` this function would not pass the ordering precondition. If this return statement was refactored to include a lambda that flipped the arguments it could pass this precondition. The result of applying this refactoring to the function from Figure 134 can be seen in Figure 135.

Finally the function `f`, after applying the lambda reordering refactoring, can be fully rewritten using the `Applicative` interface. The final result of this chain of refactorings can be seen in Figure 136.

The previous three refactorings are designed to help expand the scope over which the "generalising applicative" refactoring would work. As we said at the beginning of this section, we feel that the composite refactoring approach is better than for generalise applicative automatically to apply the refactorings described in this section. In many cases, though technically correct, the refactorings in this section can produce verbose and complicated code: for example, a lambda expression with many more arguments than two, or an inlined complex `do` expression. Producing less readable code defeats the purpose of refactoring in the first place. By taking this composite approach, a software engineer is required to be involved in each step of the refactoring, and so to be in control of the output at each step, too.

```
f = (,)
   <$> result1
   <*> (result2 <* (log <*> result3))
```

Figure 133: The inline do block can also be refactored to use the applicative interface.

```

f = do
  b <- getB
  a <- getA
  return (a,b)
```

Figure 134: The order that `a` and `b` are bound fails the precondition

The composite approach also makes it much easier for the refactoring to be extended by others. If there are other similar refactorings than we discussed here it is much simpler for another programmer to implement their own stand-alone refactoring than Figure out where in our implementation of the generalise applicative refactoring their code should be inserted.

6.6 Implementation of the Refactoring

The previous section informally discussed the "generalise monad to applicative" refactoring. In practice there are quite a few options that implementers will make when writing this refactoring, including how parentheses are inserted, for example. This section will discuss in more detail how HaRe implements the "generalise to applicative" refactoring.

The refactoring takes two arguments: the filepath of the file to be refactored, and the row and column in the file of the start of the function to be refactored. The rest of this section will go through the checking of preconditions and building the applicative expression more thoroughly. The syntax of the target function is defined in Figure 137.

```
f = do
  b <- getB
  a <- getA
  return (\ b a -> (a,b)) b a
```

Figure 135: The lambda expression in the `return` statement allows this function to pass the preconditions.

```
f = (\ b a -> (a,b)) <$> getB <*> getA
```

Figure 136: The final `Applicative` implementation of `f`.

6.6.1 Checking the Preconditions

The two arguments that the refactoring takes are enough to identify and extract the target function so that precondition checking can begin. These preconditions are defined in Figure 138.

The first precondition, checking whether any of the variables bound within the do block is used in another statement, is fairly straightforward to verify: the refactoring recurses through the list of statements and checks the right hand side of each statement to see whether a bound variable is used in it. If one of the bound variables is found in a right hand side expression then the entire do expression is not context-free and so cannot be rewritten using the applicative interface alone.

The second precondition is checked using a similar method. First we construct a list of the variables in the order that they are used in the return statement. Then we search through the `do` statements in order and whenever a binding statement is found we check that the variable being bound is the leftmost variable in the return statement that has not already been processed.

6.6.2 Constructing the Effectful Expression

Once the preconditions have passed we can start constructing the "applicative chain" that will make up the right hand side of the refactored function. This expression is

Expressions

$$
\begin{array}{lll}
e \in Expr & ::= & v \qquad\qquad\qquad Variable \\
& | & e_1 e_2 \\
& | & \lambda p \to e \\
& | & (e_1, \ldots, e_n) \qquad\qquad n \geq 2 \\
& | & \textbf{do}\ l\ e \\
& | & \ldots
\end{array}
$$

Patterns

$$
\begin{array}{lll}
p \in Pat & ::= & v \\
& | & (p_1, \ldots, p_n) \qquad\qquad n \geq 2 \\
& | & \ldots
\end{array}
$$

Statement Sequences

$$
l \in Stmts \ ::= \ \{s_1, \ldots, s_n\} \qquad n \geq 1
$$

Statements

$$
\begin{array}{lll}
s \in Stmt & ::= & p \leftarrow e \\
& | & e
\end{array}
$$

Figure 137: Input syntax for the refactoring

composed of two different types of subexpressions, binding expressions that contribute to the returned value of the function and effectful expressions that are just used for their effects on the applicative context. The code that constructs this effectful part of the applicative chain is given in Figure 139.

In the `objEntry` parser in Figure 140 only lines four and eight contribute to the final result. The other expressions just consume input and the returned values of those combinators are discarded. The first step in constructing the applicative chain involves building clusters of statements where each cluster contains a single binding statement (those matching $(p \leftarrow e) :: Stmt$) and its nearest effectful statements (those matching $e :: Stmt$). In the example we would create two clusters, the first cluster consists of lines three through six and the second of lines seven through nine.[10]

Within each cluster we then decide which applicative operator goes between each

[10]When there are an odd number of effectful expressions between binding expressions the extra expression is added to the leftmost cluster.

let $s_1, \ldots, s_n$ be the do block to refactor
s_n is the return statement.
This block must pass the following preconditions

Bound Variables are only used in return statement
$\forall x.1 \leq x < n.$
$\quad bv\ s_x \cap fv\{s_y |\ y \in \{1, \ldots, n-1\}, y \neq x\} = \emptyset$

The order variables are bound in
and appear in the return statement are the same
let $rv_1, \ldots, rv_m$ *be the variables bound in s as they appear*
$\quad$ *from left to right in the return statement* **in**
$\forall i.1 \leq i \leq m.$
$\quad \{bNum\ rv_i < bNum\ rv_j \mid j \in \{i+1, \ldots, m\}\}$

bNum returns the index of the statement
in which a variable is bound
$bNum\ v\ \{s_1, \ldots, s_n\} = y$
$\quad$ **where** $v \in bv\ s_y$
$bv\ \{s_1, \ldots, s_n\}$ = *the bound variables of* $\{s_1, \ldots, s_n\}$
$fv\ \{s_1, \ldots, s_n\}$ = *the free variables of* $\{s_1, \ldots, s_n\}$

Figure 138: Preconditions

expression. Since there is only one binding statement per cluster we know that the only options are `(*>)` or `(<*)`. We simply go through the list and if a statement occurs before the binding statement we know that it will be followed by a `(*>)` operator and once the binding statement occurs all the remaining expressions are composed with the `(<*)` operator. The first cluster from the `objEntry` example is: `(spaces *> parseStr <* spaces <* char ':')`

Once every cluster has been refactored separately all of the clusters are wrapped in parentheses and then composed with the full apply operator (<*>), as seen in Figure 141,

$$
\begin{aligned}
&buildEffects :: Stmts \rightarrow Expr \\
&buildEffects\ stmts = (e_1 <*> e_2 <*> \ldots e_m) \\
&\quad \textbf{where } \{e_1, \ldots, e_m\} = map\ buildSingleExpr\ clusters \\
&\qquad\qquad clusters = clusterStmts\ stmts \\
\\
&--BuildSingleExpr\ takes\ in\ Stmts\ that\ contains\ a\ single \\
&--bind\ Stmt\ and\ makes\ an\ applicative\ expression \\
&buildSingleExpr :: Stmts -> Expr \\
&buildSingleExpr\{bf_1, \ldots, bf_n, (p \leftarrow e), af_1, \ldots, af_m\} = \\
&\quad (bf_1 *> \ldots *> bf_n * > e <* af_1 <* \ldots <* af_m) \\
\\
&--clusterStmts\ segments\ statements\ into\ multiple\ sets\ of \\
&--statements\ each\ set\ contains\ one\ bind\ statement \\
&--and\ its\ surrounding\ statements \\
&clusterStmts :: Stmts \rightarrow \{Stmts\} \\
&clusterStmts\ \{s_1, \ldots, s_n\} = map\ (\lambda is \rightarrow map\ (\lambda i \rightarrow s_i))\ cs \\
&\quad \textbf{where} \\
&\qquad indices = \{i | s_i \in \{s_1, \ldots, s_n\}, s_i = (p \leftarrow e)\} \\
&\qquad cs = cluster\ indices\ n\ 0 \\
&\qquad cluster\ \{i\}\ l\ c = \{\{c, \ldots, (l-1)\}\} \\
&\qquad cluster\ \{i_1, i_2, \ldots\}\ l\ c = \textbf{let } b = i_1 + ((i_2 - i_1)\ `div`\ 2)\ \textbf{in} \\
&\qquad\quad \{c, \ldots, b\} : (cluster\ \{i_2, \ldots\}\ l\ (b+1))
\end{aligned}
$$

Figure 139: Building the effects

6.6.3 Building the Pure Expression

Now that the effectful part of the chain has been constructed we must build the pure expression that will be attached to the front of the applicative chain. The definition in Figure 142 builds this pure expression.

In the running example the returned values are wrapped in a tuple, which is a special case, and the pair constructor (`(,)`) will be added to the front of the chain with the infix `fmap` operator (`<$>`).

If the pure expression is a function call such as in `zipperM` from Figure 143, then

```
objEntry :: CharParser () (String, String)
objEntry = do
  spaces
  str <- parseStr
  spaces
  char ':'
  spaces
  i <- many1 digit
  spaces
  return (str, i)
```

Figure 140: The `objEntry` parser

```
   (spaces *> parseStr <* spaces <* char ':')
<*> (spaces *> many1 digit <* spaces)
```

Figure 141: The effectful expressions for the refactored definition of `objEntry`

the pure part of the expression is built by extracting the expression being returned and removing all of the bound variables from this expression. In the `zipperM` example this is `zip lst` after removing the `lst2` which was bound in the `do` block. Finally this expression is added to the front of the chain.

Once the pure expression has been added to the front of the applicative chain the new function definition can replace the right hand side of the source function. The refactored versions of both examples used in this section are shown in Figure 144.

6.7 Case Studies

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. Secondly, a good candidate will have a large corpus of code that is already written in the monadic style. If a particular library is already defined using applicative functors rather than monads then there is little work for the refactoring to do. This section will discuss some libraries where this refactoring could be useful.

```
--This takes the expression being returned
--originally and the statements and returns the expression
--to go on the front of the applicative chain
buildPureExpr :: Expr → Stmts → Maybe Expr
buildPureExpr (e1,...,en) _ = Just (,^n)
buildPureExpr e stmts = removeBoundVars e (bv stmts)

removeBoundVars :: Expr → Variable → Maybe Expr
removeBoundVars v vars =
  | v ∈ vars = Nothing
  | otherwise = Just v
removeBoundVars (e v) vars =
  | v ∈ vars = removeBoundVars e vars >>=
        (λ e' → Just e')
  | otherwise = removeBoundVars e vars >>=
        (λ e' → Just (e' v))
removeBoundVars (e1 e2) vars = do
        e1' ← removeBoundVars e1 vars
        e2' ← removeBoundVars e2 vars
        return (e1' e2')
removeBoundVars (λp → e) vars =
  | (fv e) ∩ vars ≠ ∅ = Nothing
  | otherwise = Just (λp → e)
removeBoundVars (do l e) vars =
  | (fv l) ∩ vars ≠ ∅ = Nothing
  | (fv e) ∩ vars ≠ ∅ = Nothing
  | otherwise = Just (do l e)
removeBoundVars _ _ = Nothing
```

Figure 142: Building the pure expression

6.7.1 Parsing

Many of the examples in this paper are parsers using parser combinator libraries. This has been a classic domain of applicative functors. The first examples of applicative-like

```
zipperM :: [a] -> IO [(a, b)]
zipperM lst = do
  lst2 <- getOtherList
  return $ zip lst lst2
```

Figure 143: The `zipperM` function

```
objEntry :: CharParser () (String, String)
objEntry = (,) <$>
     (spaces *> parseStr <* spaces <* char ':')
  <*> (spaces *> many1 digit <* spaces)

zipperM :: [a] -> IO [(a, b)]
zipperM lst = zip lst <$> getOtherList
```

Figure 144: The refactored functions from the two examples in this section

developments come from papers on parsing (Röjemo 1995; Swierstra and Duponcheel 1996).

Indeed, when looking through projects on Hackage that are labeled as parsers, examples of where our refactoring could be applied are numerous. Html-tokenizer is a project that tokenizes HTML code to provide a base for HTML parsers (Volkov 2016). Within it there were several functions that are can be generalised. These functions look very much like what we have already seen before. Take the example of matching an opening tag: the original definition is in Figure 145 and the same function after it has been refactored is shown in Figure 146.

Another parser that has several functions that can be refactored is the implementation of pi-forall, a simple dependently typed language created by Stephanie Weirich (Weirich 2016). The function that parses a module definition and its refactored equivalent are shown in Figures 147 and 148.

```
openingTag :: Parser OpeningTag
openingTag =
  do
    char '<'
    skipSpace
    theIdentifier <- identifier
    attributes <- many $ space *> skipSpace *> attribute
    skipSpace
    closed <- convert <$> optional (char '/')
    char '>'
    return (theIdentifier, attributes, closed)
```

Figure 145: The `openingTag` function.

```
openingTag :: Parser OpeningTag
openingTag = (,,)
  <$> (char '<' *> skipSpace *> identifier)
  <*> ((many $ space *> skipSpace *> attribute) <* skipSpace)
  <*> ((convert <$> optional (char '/')) <* char '>')
```

Figure 146: `openingTag` refactored

6.7.2 Data Store Access

One of the main motivations behind adopting the applicative interface rather than using the monadic interface is that the applicative operators are not inherently sequential: the left and right hand sides of a call to apply could be evaluated in parallel.

Facebook's Haxl project simplifies accessing remote data stores such as databases and web-services (Marlow et al. 2014). In the background when requests to the data store are composed with the `(<*>)` operator, concurrency is implicit (Marlow et al. 2014). With the generalise applicative refactoring programmers will be able to write their data queries in the do notation that they are familiar with and then refactor their code to gain the concurrency benefits. Haxl is the motivating behind the *applicativeDo* project (Marlow et al. 2016), as discussed earlier in Chapter 2.

```
moduleDef :: LParser Module
moduleDef = do
  reserved "module"
  modName <- identifier
  reserved "where"
  imports <- layout importDef (return ())
  decls <- layout decl (return ())
  cnames <- get
  return $ Module modName imports decls cnames
```

Figure 147: pi-forall's module definition parser.

```
moduleDef :: LParser Module
moduleDef = Module
   <$> (reserved "module" *> identifier <* reserved "where")
   <*> layout importDef (return ())
   <*> layout decl (return ())
   <*> get
```

Figure 148: The refactored module definition parser

6.7.3 Yesod

Another possible application of this refactoring is to parts of the code used to define Yesod webservers (Snoyman 2016). The preferred way to handle the creation and processing of web forms uses 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 the more familiar do notation and then refactor their code to fit the community standard.

6.8 Summary

This refactoring is another example of a generalisation refactoring that rewrites code to work with a type class. The previous examples of this type of refactoring, the Maybe to MonadPlus or the Maybe to Monad refactorings generalised a concrete type to one

of its type classes. This refactoring is different, it does not generalise a program so that additional concrete types can be applied to it. This refactoring generalises the way that the target program can be evaluated. Though monads can be constructed to support parallel evaluation (Marlow, Newton and Peyton Jones 2011), this functionality needs to be explicitally added to a program. Applicative programs can be made implicitly parallel depending on the definition of the applicative instance (Marlow et al. 2014).

Additionally there are cases where the applicative versions of functions are easier to read. Parsers in particular can become very descriptive in the applicative style with function calls being read from left to right in the same order as the syntax elements they are parsing appear in the input.

In both of these cases the type of data the program is processing will drive the desire to refactor. The first case the parts of a program's result are computed under or extracted from a monadic context but these parts are independently computed and don't depend on the result of one part to compute other parts. This opens the opportunity for parallelism by switching from the strictly sequential monadic interface to the applicative one. With some changes to how apply is evaluated, each of its arguments can be computed in parallel if the applicative instance is defined to take advantage of this fact (Marlow et al. 2014). The other case, the actual format of the data can inspire the refactoring. Haskell's `do` syntax has strict formatting requirements. Every statement of a `do` block must begin on the same column in the source file. The applicative operators are not as strict and would allow for parsing code to be formatted in unique ways to increase readability.

Improving readability is a valid reason to refactor code, but when is refactoring preferable over a compiler optimization? The `ApplicativeDo` desugarer from Marlow et al. (2016) (see Section 2.4) automatically introduces applicative operators into a `do` statement, why would a refactoring be preferable to just letting the compiler optimize your program for you? Programmers may prefer their program's semantics to be explicitly written rather than trusting the compiler to make the correct decisions.

Chapter 7

Introducing Monads

Up to this point the data refactorings that have been discussed are making changes to abstractions that already exist in the source code. This chapter explores refactorings that *introduce* effectful abstractions into pure code. In particular this chapter focuses on monadification, the process of adding monads to pure code.

Monadification of a Haskell program is a common transformation that must be performed when a program is explicitly effectful, such as using `Maybe` to handle failure or passing the state of a computation as a parameter. The Haskell Community's guidelines also advise that monadic and pure code should be separated if possible, so that Haskell programs should be written as pure code until it is clear that monads are required (HaskellWiki 2015c). This makes the process of monadification a common transformation that Haskell programmers undertake, making it a compelling refactoring target.

This chapter describes a monadification refactoring. First there is a brief review of the `Monad` type class from Haskel (section 7.1). There are multiple styles of monadification, as described in (Reinke et al. 2005), and Section 7.2 will describe these styles and discuss their relative merits and usefulness to Haskell programmers. Finally the implementation of the monadification refactoring within HaRe is described in section 7.3.

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

class Functor f => Applicative f where
  pure :: a -> f a

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

class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  m >> k = m >>= \_ -> k

  return :: a -> m a
  return = pure

  fail :: String -> m a
  fail s = errorWithoutStackTrace s
```

Figure 149: The monad type class

7.1 The Monad type class

The monad type class is defined in Figure 149 along with the `Functor` and `Applicative` type classes for reference. The two canonical monad functions are `return` and "`(>>=)`" which is pronounced bind. `return` is the function that brings some pure value into the monadic context; the default definition is the function `pure` from the `Applicative` instance of the type. Bind allows for computation to occur within the monadic context.

The other two functions are part of the Haskell language standard for the `Monad` type class. The `(>>)` operator allows for sequential composition, when the second operation is not dependent on the outcome of the first. `fail` is called when a pattern match failure occurs inside of a `do` statement. In Figure 150 `fail` would be called if the function `f` returns `Nothing` rather than (`Just n`) for some `n`.

```
1 do{
2    (Just n) <- f;
3    return (n+1)
4 }
```

Figure 150: A `do` statement with a possible call to `fail` on line 2.

Haskell has identified monads as the standard framework for building I/O and stateful computations. This section has described the monad type class. Despite the number of tutorials written about how to understand it, the monad type class only consists of the `return` and `(»=)` operations. The following section will discuss the different ways that monadic programs can be constructed.

7.2 Styles of Monadification

Up to this point this thesis has not gone into detail about what monadification is, beyond the process of introducing monads into a previously pure type. Monadifying a single type changes it from type `a` to type `Monad m => m a`. Monadifying a function will always change the result type of a target function because there is no way to extract values out of any monadic context. Monadification can also affect any combination of parameters. Which parameters are affected depends on the *style* of monadification.

The different styles of monadification can change how a function is evaluated. In Reinke et al. (2005) five different styles of monadification are described as part of a survey of the Haskell community's use of monads. The monadification styles are:

- Full Call-by-Value
- Full Call-by-Name
- Restricted Call-by-Name
- Data-Directed

```
1 mergesort :: (Ord a) => [a] -> [a]
2 mergesort lst = let (l,r) = splitAt ((length lst) `div` 2) lst
      in
3   merge (mergesort l) (mergesort r)
4
5 merge :: (Ord a) => [a] -> [a] -> [a]
6 merge [] r = r
7 merge l [] = l
8 merge (x:xs) (y:ys) =
9   case (x < y) of
10  True -> x:(merge xs (y:ys))
11  False -> y:(merge (x:xs) ys)
```

Figure 151: Mergesort

- Restricted Call-by Value (This is the style implemented in HaRe)

The rest of this section will describe each of these monadification styles in more detail. There are two examples used: one is an implementation of merge sort that was developed for this thesis, and the other is an expression evaluator which is taken from Reinke et al. (2005).

7.2.1 Full Call-by-Value Monadification

A call-by-value monadification of a function will ensure that all arguments are evaluated before being passed to a function and the function will also be evaluated before having arguments passed to it. Consider the definition of mergesort in Figure 151.

The full call-by-value monadification, when applied to `mergesort`, is defined in Figure 152

7.2.2 Full Call-by-Name Monadification

The full call-by-name monadification of a function ensures arguments are unevaluated when they are passed to the function. The full call-by-name version of `mergesort` is defined in Figure 153.

```
1
2 mergesort :: (Ord a, Monad m) => m ([a] -> m [a])
3 mergesort = return (\ lst -> case lst of
4              [] -> return []
5              [x] -> return [x]
6              _ -> do
7               let (l,r) = splitAt ((length lst) `div` 2) lst
8               mm1 <- mergesort
9               v1 <- mm1 l
10              mm2 <- mergesort
11              v2 <- mm2 r
12              return (merge v1 v2))
```

Figure 152: Full call-by-value monadification of mergesort

```
1 mergesort :: (Ord a, Monad m) => m (m [a] -> m [a])
2 mergesort = return (\ mlst -> do
3    lst <- mlst
4    res <- case lst of
5                [] -> return []
6                [x] -> return [x]
7                _ -> do
8                 let (l,r) = splitAt ((length lst) `div` 2) lst
9                 mm1 <- mergesort
10                v1 <- mm1 (return l)
11                mm2 <- mergesort
12                v2 <- mm2 (return r)
13                return (merge v1 v2)
14   return res)
```

Figure 153: Full call-by-name monadification of mergesort

The two "full" monadification styles monadify not only the arguments to a function but also the function itself. That is the refactored functions return a monadic value that computes another function with, depending on the style, parameters that may be monadic.

The full monadification styles are a bit “too monadified” to be broadly useful. Consider the full call-by-value monadification of `mergesort` in Figure 152, to sort a list, first the function needs to be extracted via the monadic context then it can be applied

```
> let lst = [3,2,1]
> fcbvMergesort >>= (\f -> f lst)
[1,2,3]
> fcbnMergesort >>= (\f -> f (return lst))
[1,2,3]
```

Figure 154: Using the monadified versions of mergsort. `fcbvMergsort` is defined in Figure 152 and `fcbnMergesort` is defined in Figure 153.

```
mergesort :: (Ord a, Monad m) => m [a] -> m [a]
mergesort ml = do
  lst <- ml
  res <- case lst of
    [] -> return []
    [x] -> return [x]
    _ -> do
      let (l,r) = splitAt ((length lst) `div` 2) lst
      v1 <- mergesort (return l)
      v2 <- mergesort (return r)
      return (merge v1 v2)
  return res
```

Figure 155: The restricted call-by-name monadification of mergesort.

to the list. The full call-by-name version of the function takes an additional step to use, this time the list value needs to be lifted into the monadic context before it can be passed to the extracted function. Examples of how to use these full monadifications are shown in Figure 154.

The remaining monadification styles are "restricted" versions of the two full monadification styles. All of these styles do not wrap the function inside of the monadic context, they differ in which of the target function's parameters are monadified.

7.2.3 Restricted Call-by-Name Monadification

This style of monadification is where every argument is contained within the monad. Using the mergesort example again, Figure 155 shows the restricted call-by-name monadification.

```
1 data Expr = Lit Int
2   | Bin Op Op Expr Expr
3
4 data Op = Add | Div
5
6 eval :: Expr -> Int
7 eval (Lit n) = n
8 eval (BinOp op e1 e2)
9   = evalOp op (eval e1) (eval e2)
10
11 evalOp :: Op -> Int -> Int -> Int
12 evalOp Add v1 v2 = v1 + v2
13 evalOp Div v1 v2 = v1 `div` v2
```

Figure 156: The pure interpreter implementation

The argument passed to `mergesort` in Figure 155 is within the monad so it will not be evaluated until it is extracted from the monadic context on line 3 of Figure 155. This gives the refactored function its call-by-name "flavour" as Reinke et al. (2005) describes it.

7.2.4 Data-Directed Monadification

This style refactors a certain type to be replaced by a monadic computation over that type. This example becomes clearer when working with functions that have more than a single argument like the `mergesort` example. This style will be explained using the example from Lämmel (1999). The example is a simple arithmetic expression interpreter as seen in Figure 156.

With this particular example the programmer wants the expressions to be evaluated within a monad to add effects. Since `Int` represents the result of an expression in this program this type is monadified as seen in Figure 157. Importantly this refactoring makes the type of `evalOp` change to `Monad m => Op -> m Int -> m Int -> m Int` in addition to monadifying the result type of `eval`.

```
eval :: Monad m => Expr -> m Int
eval (Lit n) = return n
eval (BinOp op e1 e2)
  = evalOp op (eval e1) (eval e2)

evalOp :: Monad m => Op -> m Int -> m Int -> m Int
evalOp Add v1 v2 = do
  l <- v1
  r <- v2
  return (l + r)
evalOp Div v1 v2 = do
  l <- v1
  r <- v2
  return (l `div` r)
```

Figure 157: A data-directed monadification of the interpreter.

Some arguments are passed unevaluated- those of monadic type- whereas arguments of other types are non-monadic. Therefore this style is a mixture of the call-by-name and call-by-value styles (Reinke et al. 2005).

Data-directed monadification is useful if a particular type can always represent the result of a computation that should be effectful. This makes it a difficult style of monadification to implement, because there is not enough information in the type system to always determine which parameters are targets of the refacoring. In the interpreter case, `Int` is always the result of an evaluation. Even in the only call to `evalOp` the second and third arguments are computed from recursive calls to `eval` so those `Int` types also represent expression evaluation. This style of monadification would pair well with the "introduce a type synonym" refactoring to differentiate instances of types that should be monadified.

7.2.5 Restricted Call-by-Value Monadification

The final monadification style monadifies the result type of the target function(s). This is the style of monadification that is produced by the algorithm described in Erwig and

```
mergesort :: (Ord a, Monad m) => [a] -> m [a]
mergesort lst = do
  let (l,r) = splitAt ((length lst) `div` 2) lst
  v1 <- mergesort l
  v2 <- mergesort r
  merge v1 v2

merge :: (Ord a, Monad m) => [a] -> [a] -> m [a]
merge [] r = return r
merge l [] = return l
merge (x:xs) (y:ys) =
  case (x < y) of
  True -> do
    rst <- (merge xs (y:ys))
    return (x:rst)
  False -> do
    rst <- (merge (x:xs) ys)
    return (y:rst)
```

Figure 158: Restricted call-by-value monadification

Ren (2004) and that is implemented in HaRe as part of this thesis work.

Returning to the mergesort example that was used earlier in this section, the restricted call-by-value monadification implementation of this can be seen in Figure 158.

This style of monadification meshes nicely with the bind operator which expects functions whose arguments are pure values and returns a monadic type. This style of monadification has been implemented in HaRe and the next section will go into greater detail about the implementation of this refactoring.

This section has been a summary of the different ways that monadic programs can be written. These styles were first identified by Lämmel (1999), Erwig and Ren (2004), and Reinke et al. (2005). This thesis provides an implementation of the restricted call-by-value monadification. This style fits well with how the passed to bind is a pure value and it returns a monadic value. This style is also the obvious way to make a non-effectful computation effectful. The other styles assume that there is an already existing monadic context that either produces the arguments that are passed to the target

function, in the call-by-name cases, or to extract the function into, as in the "full" monadification styles. The restricted call-by-value style of monadification fits easily into an otherwise pure program, which is what the refactoring is targeting.

7.3 Implementation of the Monadification Refactoring

The implementation of Monadification in HaRe is a top down transformation. The monadification refactoring takes a list of target functions as input. In practice the refactoring receives a list of positions in a file and these positions indicate which functions should be targeted by the refactoring.

The monadification refactoring works over sets of functions simultaneously. Figure 159 defines two functions `f` and `g` and the two possible monadifications ("`*_m1`" and "`*_m2`"). The `f_m1` function is how `f` would be rewritten if it was the sole target of the refactoring, and `f_m2` and `g_m2` are the definitions if both functions were a target of the refactoring.

When only making `f` effectful the refactoring can simply lift its original definition into the monad using the `return` function. Since `g` is a pure function it can still be used in the body of `f_m1` without any changes. When both of the functions are targets of the refactoring `g`, because it is not dependent on any other monadic functions, can just be handled in the same way `f_m1` was by lifting its definition into the monadic context using `return`. However, `f` is dependent on the now effectful `g` so the expression "`(g x) + 1`" will now throw a type error because `(g x)` is of type "`m Int`" not a number which was it's previous type. Instead the call to `g_m2` needs to be bound to the rest of `f_m2`'s definition so that its result can be extracted from the monadic context.

It's important to note that `g` alone is not a valid target of the refactoring. This is because `f` is dependent on `g`, if `g` becomes effectful then it can no longer be used the pure function `f`.

```
f :: Int -> Int
f x = (g x) + 1

g :: Int -> Int
g y = y * 2

f_m1 :: (Monad m) => Int -> m Int
f_m1 x = return ((g x) + 1)

f_m2 :: (Monad m) => Int -> m Int
f_m2 x = g_m2 x >>= (\v1 -> return (v1 + 1))

g_m2 :: (Monad m) => Int -> m Int
g_m2 y = return (y * 2)
```

Figure 159: Possible monadification refactorings for the `f` and `g` functions.

```
f :: Int -> String -> String
f n str = (take n str, drop n str)


stringHandler :: Maybe Int -> String -> (String, String)
stringHandler mi s =
  let g = case mi of
    Nothing -> f ((length s) `div` 2)
    (Just i) -> f i
  in g s
```

Figure 160: The `stringHandler` function will be rejected by the monadification refactoring.

7.3.1 Preconditions

There are two preconditions that must be met before this refactoring can be applied. First, all of call points of the target functions must be "fully-saturated." This means that every call point will have a named variable passed to the target function. This allows the refactoring to assume that the type of a subtree whose leftmost child is a monadified function is a monadic value and not something of type `Monad m => a -> m a`.

```
1 stringHandler :: Maybe Int -> String -> (String, String)
2 stringHandler mi s =
3   let g = case mi of
4     Nothing -> (\x -> f ((length s) `div` 2) x)
5     (Just i) -> (\x -> f i x)
6   in g s
```

Figure 161: The new version of `stringHandler` can be monadified.

Like the many of the preconditions for the "generalise monad to applicative" refactoring, other refactorings can help transform programs to pass the preconditions. The `stringHandler` in Figure 160 has calls to `f` that are not fully saturated. Fortunately η-expansion can be used to fully saturate the calls to `f` (see Figure 161) allowing `stringHandler` and `f` to be monadified.[1] This expansion involves introducing abstraction over a function so given some function: `f :: a -> b -> c` which is used in the expression: `f x`, this can be η-expanded to: `(\y -> f x y)` so now all of `f`'s parameters at this call site are named variables.

The second precondition of this refactoring is that the target functions cannot be called outside the scope of the refactoring. Once a function has been monadified it's result cannot be used within a pure function any longer because there is no way to retrieve the result of a monadic computation from the monadic context.

In the `stringHandler` example from Figure 161, `stringHandler` alone and `stringHandler` and `f` are valid target sets of the refactoring, but trying to monadify `f` alone is not because it is used outside the scope of the refactoring.

7.3.2 The Transformation

The simplest case of this refactoring is when the target function is defined using only pure functions then the refactored version of the function is simply `return` applied to the original body of the function. Otherwise the refactoring works over the body of a target function in four steps:

[1] The original definition of `stringHandler` could be monadified alone.

1. The refactoring begins by extracting any sub-expressions that are calls to a target function. The extraction happens in a top-down left to right manner, when a sub-expression is found that is a call to one of the refactoring's target functions it is replaced with a variable, so `(f x = f (g x))` would become `(f v1)`.

2. Next a bind expression needs to be constructed from the extracted monadic sub-expressions and the expressions they were extracted from. The extracted sub-expressions become the first argument to bind and a newly constructed lambda expression makes up the second argument. The lambda expression is made from the expression that had the monadic expression replaced with some variable, the only argument of the lambda expression is that variable. From the example in the previous step: `(g x)` is the extracted monadic sub-expression and `(f v1)` is the expression it was replaced with the `v1` variable. In this example the constructed bind expression is defined as `(g x) >>= (\v1 -> f v1)`.

3. If the right hand side expressions of one of the lambda expressions is not a monadic value then `return` will have to be applied to this expression. In the example from the previous step, if `f` is a target of the refactoring then the bind statement does not need to be modified further. If `f` is not one of the target functions then the bind expression needs to be rewritten to: `(g x) >>= (\v1 -> return (f v1))`.

4. Finally the body of the target function can be replaced with the constructed bind expression. This target function has now been successfully refactored. These four steps need to be performed over the body of each of the target functions.

A more concrete example is shown in Figure 162. The functions `f`, `g`, and `h` are all targets of the refactoring.

The syntax tree of the right hand side of `f` is shown in Figure 163. The first step of the refactoring begins by extracting all the monadic sub-expressions of the body of `f` and replacing them with variables. This step works in a top down left to right manner so

```
1 f x y = f (h x) (g y)
```

Figure 162: A simple target function for monadification

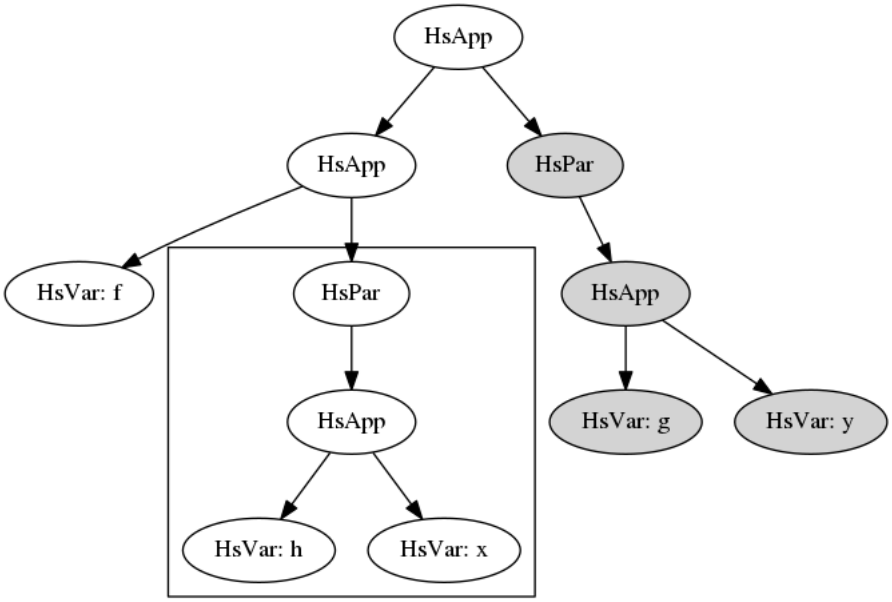


Figure 163: The syntax tree of `f` from Figure 162

the call to `h` is discovered first, the relevant sub-tree is outlined in Figure 163. Next the call to `g` will also be extracted. The implementation of the refactoring contains a queue that keeps track of the monadic expressions and the variables that replaced them. At this point in the refactoring, after step one, the queue contains two elements: `[(v1,(h x)), (v2,(g y))]` and the body of `f` has become: `f v1 v2`.

Next in step two the bind expression will be constructed from the contents of the queue and the rewritten body of `f`. The refactoring begins by popping each expression off of the queue, composing it with bind and creating a lambda expression from the corresponding variable name. This process has been illustrated in Figure 164.

Once the bind expression has been produced as shown in line 9 of Figure 164 the third step of the refactoring checks if its innermost expression needs to be lifted into the monadic context with `return`. In this particular case since the innermost expression is

```
-- Initial state of the queue: [(v1,(h x)), (v2,(g y))]
-- Begin by popping (v1, (h x)) off
(h x) >>= (\ v1 -> ...)
-- The right hand side of the lambda is constructed from what remains in the queue
-- Pop (v2, (g y)) off
(h x) >>= (\ v1 -> (g y) >>= (\ v2 -> ...))
-- The queue is now empty so the top level expression is inserted here
-- The final result of step 2:
(h x) >>= (\ v1 -> (g y) >>= (\ v2 -> f v1 v2))
```

Figure 164: Working through the queue of monadic expressions.

```
f x y = (h x) >>= (\ v1 -> (g y) >>= (\v2 -> f v1 v2))
```

Figure 165: The final refactored version of `f`

a call to a target function so this step does not affect the final product of the refactoring.

Not the binding of `f` can be replaced with the bind expression. The final result of refactoring `f` is shown in Figure 165. After rewriting `f` the refactoring would continue by refactoring the bodies of `g` and `h`

At this point the refactoring is capable of transforming simple function application. Haskell, obviously, has more expressive expressions beyond just function applications. The next section will describe how this implementation handles let expressions.

7.4 Let Expressions

The previous section details how the refactoring works over simple function applications. This section will describe how a function with a let expression is refactored. This section will use the expression evaluator in Figure 166 as a motivating example.

The idea is that at this point in the project the explicit passing of the state is unwanted so refactoring `eval` to become monadic so that the `Env` can be passed around using the `State` monad instead. The first two cases that handle the `Var` and `N` constructors can simply be wrapped with `return` because no recursive calls to `eval`

```
1  data Expr = Var Char
2            | N Int
3            | Add Expr Expr
4            | Sub Expr Expr
5            | Assign Char Expr
6           deriving Show
7
8  type Env = [(Char,Int)]
9
10 eval :: Expr -> Env -> (Int,Env)
11
12 eval (Var v) env = (head [val | (x,val) <- env, x==v], env)
13
14 eval (N n) env = (n,env)
15
16 eval (Add e1 e2) env = let (v1,env1) = eval e1 env
17                              (v2,env2) = eval e2 env1
18                          in (v1+v2,env2)
19 eval (Sub e1 e2) env = let (v1,env1) = eval e1 env
20                              (v2,env2) = eval e2 env1
21                          in (v1-v2,env2)
22 eval (Assign x e) env = let (v,env1) = eval e env
23   in (v, (x,v):env1)
```

Figure 166: An expression evaluator.

happen in those cases. The three other cases contain let bindings that all call (the now monadic) `eval`.

In those cases the local bindings can be used as the variables inside of a lambda expression. This seems simple enough, there is a question of the order in which the bindings should be processed? Consider the let binding in Figure 167, assuming that the `f` and `x` variables are in scope. Due to lazy evaluation this is a valid Haskell let binding despite `v1` being referenced on line one before it is defined on line two. However if the let bindings are processed in the obvious way from top to bottom[2] `v1` will not be in scope when it is passed to `g`.

Fortunately GHC offers a solution: the renamer of GHC, which checks for lexical

[2]In this case: `g v1 >>= (\v2 -> f x >>= (\v1 -> ...))`

```
1 let v2 = g v1
2     v1 = f x
3    in ...
```

Figure 167: An interesting let binding

```
1  eval :: Monad m => Expr -> Env -> m (Int,Env)
2
3  eval (Var v) env = return (head [val | (x,val) <- env, x==v],
       env)
4
5  eval (N n) env = return (n,env)
6
7  eval (Add e1 e2) env = eval e1 env >>=
8    (\(v1,env1) -> eval e2 env1 >>=
9      (\(v2,env2) -> return (v1+v2,env2)))
10
11 eval (Sub e1 e2) env = eval e1 env >>=
12   (\(v1,env1) -> eval e2 env1 >>=
13     (\(v2,env2) -> return (v1-v2,env2)))
14
15 eval (Assign x e) env = eval e env >>=
16   (\(v,env1) -> return (v, (x,v):env1)))
```

Figure 168: The monadified expression evaluator.

errors, also performs dependency analysis on local bindings and orders them so that later bindings may depend on earlier ones but not vice versa when there are no mutually recursive bindings (GHC API 2016). This renamed ordering is used to lookup the parsed bindings in dependency analysed order.

Returning to the evaluator example the monadified version is shown in Figure 168.[3] The let bindings in `eval` can be processed from top to bottom. In the `Add` case this means that the expression bound on line 16 in Figure 166 makes up the outermost call to bind and the pattern it was originally bound to becomes the pattern that matches the argument passed to the first lambda expression.

[3]Line breaks have been added to aid readability in this setting but the refactoring would not actually do this.

```
1 f x = let y = pureF x
2           z = monF y
3       in z+x
4
5 f_m x = let y = pureF x in
6    monF y >>= (\z -> return z+x)
```

Figure 169: A function with a mixture of pure and monadic function calls in a let expression.

If there were pure expressions on the right hand side of the let binding those bindings will remain in a let expression. For example the function starting on line one in Figure 169 contains two let bindings `pureF` is a pure function and so it remains inside a let expression whereas `monF` is a target of the refactoring and it is lifted from the let and passed to bind.

With this section the refactoring supports the transformation of the core of Haskell, the the typed lambda calculus with polymorphic let expressions. However, this refactoring outputs monadic code composed with the bind operator, which is probably not the most popular way to write monadic code. To better support monadic programming Haskell gives monads their own syntactic structure known as do-syntax. The next section will discuss a separate refactoring for introducing this syntax into the target function.

7.5 Adding Syntactic Sugar

The refactored program from Figure 168 is rewritten in a restricted call-by-value monadification style. However, idiomatic monadic Haskell is not often written in this style. Haskell also supports a piece of syntactic sugar called `do` notation. This notation allows for a series of bind computations to be written in a more compact and easier to read style. Figure 170 shows how the bind expression on line one can be sugared into the do statement on lines four through six.

```
m_expr >>= (\x -> return (f x))
--Sugars to

do
  x <- m_expr
  return (f x)
```

Figure 170: An example of how binds can sugar to `do` notation.

do notation is highly used when writing monadic Haskell programs and some texts even introduce it before explaining what monads are (O'Sullivan, Goerzen and Stewart (2008); Thompson (2011); Lipovača (2012)). Supporting this syntax is an important feature for a monadification refactoring to be practical.

The sugaring of bind syntax into `do` syntax is a separate refactoring in HaRe. This allows for the programmer to choose exactly which functions should be written with `do` syntax and when functions are clearer remaining as binds. The refactoring takes in a single parameter, the position where the target function is declared. The sugared version of `eval` from Figure 168 is shown in Figure 171. The first two cases weren't bind expressions so they remain just calls to return. The three other cases have been refactored into the equivalent `do` statements.

Introducing the do syntactic sugar into the target program is the final step of this two-step composite refactoring. Do syntax has become synonymous with monads in the Haskell language and a refactoring that supports them should also support this syntax as well.

7.6 Summary

Monads are a challenging and characteristic feature of Haskell and, anecdotally at least, infamous amongst those trying to learn the language. Monadification is an important and common transformation that Haskell code undergoes. This chapter has described two refactorings that automate this transformation. The monadification refactoring

```
1 eval :: Monad m => Expr -> Env -> m (Int,Env)
2
3 eval (Var v) env = return (head [val | (x,val) <- env, x==v],
    env)
4
5 eval (N n) env = return (n,env)
6
7 eval (Add e1 e2) env = do
8   (v1,env1) <- eval e1 env
9   (v2,env2) <- eval e2 env1
10  return (v1+v2,env2)
11
12 eval (Sub e1 e2) env = do
13  (v1,env1) <- eval e1 env
14  (v2,env2) <- eval e2 env1
15  return (v1-v2,env2)
16
17 eval (Assign x e) env = do
18  (v,env1) <- eval e env
19  return (v, (x,v):env1))
```

Figure 171: The sugared evaluator.

transforms a set of functions into restricted call-by-value monad style, where only the result type of a function is made monadic. The second refactoring sugars the binds produced by the monadification into `do` notation, a common structure for monadic Haskell functions.

Unlike the other refactorings developed for this thesis this transformation has a long history in the Haskell programming community. The design space for this refactoring had already been extensively covered in (Lämmel (1999); Erwig and Ren (2004); Renke (2005)). Rather than producing an additional style of monadification the contribution of this chapter is the implementation of a practical monadification refactoring based on the styles of monadification, from Renke (2005), and algorithms for introducing them provided by Lämmel (1999) and Erwig and Ren (2004).

Chapter 8

Conclusion

This thesis has explored a new category of refactoring for functional programming languages. Data-driven refactorings are transformations that are *driven* by the data types a program uses. The generalisation refactorings, for example, are motivated by a desire to allow code to be re-used in more places (e.g. the "Maybe to MonadPlus" refactoring) or even to generalise the way a program is evaluated as in the "monad to applicative" refactoring where the `Applicative` type class does not force sequential evaluation of the arguments to apply (<*>) like bind (»=), the monad operation, does.

These refactorings are all designed to help programmers redesign the data their programs use and manipulate. It has been shown in the literature that as programmers make structural decisions the implementation of their programs technical debt can build up, and this debt can only be paid through rewriting and/or refactoring (Cunningham (1992); Fowler (1999)). The core idea behind this thesis is that the data representation decisions that are also made during program development also accrue debt, and that refactoring is a valid and simple way to pay off this type of technical debt.

8.1 Summary of Contributions

The primary artifacts of this thesis are my contributions to HaRe and its API. In particular the following contributions have been made:

- The design and implementation of a set of data-driven refactorings in HaRe. These refactorings are:
  - Introduce a Type Synonym, Section 4.3
  - Generalise Maybe to MonadPlus/Monad, Section 4.4.2 and Chapter 5
  - List to Hughes List, Section 4.5 and chapter 5
  - Generalise Monad to Applicative, Chapter 6
- The implementation of Monadification, Chapter 7
- An API for the creation of embeddable types refactorings, chapter 5
- Enhancements and additions to HaRe's API, see Chapter 5. The API was extended while developing the refactorings mentioned above.
  - Functions that perform high level, common transformations, that many refactorings must do (e.g. wrapping a syntax element in parentheses)
  - Functions for retrieving particular syntax elements from the abstract syntax tree of an entire module such as getting the body of a function based on a position.
  - Functions that aid in working with ghc-exactprint and the parsed GHC abstract syntax tree's annotations. These functions provide functionality such as changing the location of a syntax element relative to the prior syntax element or adding new annotations to a syntax element.

8.2 The GHC Tooling Ecosystem

The focus of the refactorings described in this theis are the changes they make to the data that the target programs use; any structural changes that occur during the transformation are incidental. Despite not being the focus of this thesis, a great deal of effort had to go into understanding the state of the GHC tooling ecosystem and how to rewrite GHC Haskell to produce it. This section will summarize my experience with the current state of the program transformation ecosystem of GHC.

I began working on HaRe when the latest version of GHC was 7.6 (At the time of writing it is 8.4.3). At this point all comments, whitespace information, and the positioning of many special characters were completely eliminated from the parsed abstract syntax. To preserve the layout of source files HaRe had to generate a separate tree structure that contained all the tokens that were not represented by the AST and some whitespace information. When it came time to print the modified source the AST and this token tree needed to be combined for the output to be formatted correctly.

A major addition to the utility of the GHC API came with the release of GHC 7.10. With this release the GHC parser would now return the location of all comments and tokens that had been previously discarded, in a separate structure as part of the the parsed syntax tree. This implementation allowed compiler stages that depended on the parsed AST to remain unchanged because the new structure could be ignored. This addition to the GHC also coincided with the release of ghc-exactprint,[1] which was described in Section 3.4. This library made transforming Haskell code much easier because it allowed the formatting to be done with relative positioning instead of absolute locations. Ghc-exactprint also greatly simplified the development of the common transformation library that was added to HaRe as a part of this thesis work.

In the short time that I have been a part of the Haskell tooling community it has

[1] This is not coincidental. Alan Zimmerman is the programmer behind both the modification to the GHC and the ghc-exactprint library.

grown and changed a great deal. The GHC has been changed specifically to better support tool builders, notable examples of this are Zimmerman (2015) and Najd and Jones (2017). There is also an initiative to create a pluginable editor interface for Haskell, haskell-ide-engine[2] so that a Haskell environment can be easily incorporated into many different editors and IDEs and tool developers only have to target the ide engine plugin system. The future of Haskell tooling is looking very bright! And in particular this makes adding HaRe to other IDEs that much easier.

8.2.1 Challenges of Working with the GHC API

Working with the GHC API is not without its challenges however. The structure of the abstract syntax is very complex and spread across several modules. The core types of the tree are represented by many constructors, the expression type for example, `HsExpr`, contains 51 different constructors as of GHC 8. This amount of complexity makes a generic programming library a mandatory prerequisite for working with the abstract syntax tree.

Fortunately for tool builders the number of types and constructors that represent the vast majority of "standard" Haskell code is much more limited. However, determining which constructors you need to target for any given transformation can be very tedious: in the author's experience this process involves looking at numerous printouts of abstract syntax trees.

This situation could be improved by centralising information for GHC tool builders. Currently the Haskell wiki hosts a single page "GHC/As a library" that gives a few examples of how to run the various stages over a single file (HaskellWiki 2017a). This is a fine general introduction to working with the GHC API but it does not contain many details about working with the abstract syntax or projects. The GHC developer wiki[3] does contain a great deal of detail about the inner workings of but it is not well

[2] `https://github.com/haskell/haskell-ide-engine`
[3] `https://ghc.haskell.org/trac/ghc/`

organised or easy to find, and a lot of the material there is not directly relevent to tool builders. A centeralised source of Haskell tool building information would help guide people new to the field greatly.

8.3 Future Work

This work and HaRe can be extended in multiple ways. Development on HaRe is ongoing and will continue for the foreseeable future.

Reimplement Refactorings - When HaRe was updated to work with GHC the original implementations of the refactorings were no longer valid using the new back-end. An important task in the near future will be to go back an reimplement the refactorings that are currently not supported in HaRe.

Template language for refactoring - A powerful feature of Wrangler is its template language (Li and Thompson 2012). This allows for Wrangler refactorings to be defined using concrete Erlang syntax rather than the abstract syntax, lowering the difficulty in implementing refactorings significantly for programmers not familiar with the Erlang backend. A similar feature for HaRe would allow refactorings to be written without having to understand the syntax tree of GHC. Template Haskell is a template language for Haskell and is built into the GHC. A significant amount of time while developing this thesis was spent exploring if Template Haskell and its related feature, Quasiquotation (Mainland 2007), could be used to develop a template language for refactoring Haskell. Unfortunately Template Haskell uses its own abstract syntax for Haskell code which makes it difficult to use for refactoring GHC Haskell. However, there is a plan to change this though it is not finished yet (GHC Developer Wiki 2017).

Data-Driven refactorings for other languages - The data-driven refactorings presented in this thesis lean heavily on the types that Haskell provides and how they are

used by the Haskell community. Data-driven refactorings for other programming languages would be very different from the refactorings presented in this thesis. The refactorings related to Applicative Functors and Monads are much less useful for other langauges as they are a fairly unique feature of Haskell.[4] Data-driven refactorings for other statically typed functional languages such as OCaml or Scala would be an interesting topic to explore further because of how different these langauges are from Haskell.

Interactive refactorings - Many of the more complex refactorings would benefit from becoming more interactive. For example after the "introduce type synonym" refactoring completes HaRe could highlight instances of the target type and the user could indicate if each instance should be replaced by the newly introduced synonym. Another possible interactive feature of a refactoring would be to offer to refactor some code if it failed the preconditions for another refactoring. For example, if when trying to refactor a `do` block to use applicative operations the block fails the precondition requiring that all left hand side variables not be used in a right hand side expression, HaRe could suggest extracting the monadic code into its own function and then performing the original refactoring on the modified function.

The feature also deals with one of the major challenges that refactoring tools deal with when they try and attract users; a refactoring tool is unable to produce the exact target code that the user is looking for. Potential users may dismiss a tool that produces code that is 90% of what they want because getting the outputted code the way they want it may take just as much work as refactoring the code manually. If a refactoring tool can automatically produce most of the target code then ask the right questions to produce the rest of the code the tool becomes much more usable.

[4]Haskell's *purity* requires some way to handle effects and monads are the primary way this is handled in Haskell. However, nothing prevents monads from being adopted in any language that supports parameterised types they are just less useful in an effectful language.

Bibliography

(2015). Erlang. http://www.erlang.org/, [Online; accessed 13-April-2015].

Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors*. Ph.D. thesis, Mikroelektronik och informationsteknik.

Armstrong, J. (2007). A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, ACM, pp. 6–1.

Bajt, A. (2014). 24 Days of GHC Extensions: Deriving. https://ocharles.org.uk/blog/guest-posts/2014-12-15-deriving.html, [Online; accessed 27-May-2018].

Beck, K. (2000). *Extreme programming explained: embrace change*. addison-wesley professional.

Brown, C., Loidl, H.-W. and Hammond, K. (2011). Paraforming: forming parallel haskell programs using novel refactoring techniques. In *International Symposium on Trends in Functional Programming*, Springer, pp. 82–97.

Brown, C. et al. (2014). Cost-directed refactoring for parallel erlang programs. *International Journal of Parallel Programming*, 42(4), pp. 564–582.

Brown, C. M. (2008). *Tool support for refactoring haskell programs*. Ph.D. thesis, University of Kent.

Burazin, I. (2014). Most Popular Desktop IDEs & Code Editors in 2014. `https://blog.codeanywhere.com/most-popular-ides-code-editors/`, [Online; accessed 01-August-2017].

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1), pp. 44–67.

Carlsson, R. (2015). Erlang Syntax Tools. `http://erlang.org/documentation/doc-5.10.2//lib/syntax_tools-1.6.11/doc/html/chapter.html`, [Online; accessed 27-May-2018].

clojure emacs (2018). clj-refactor.el. `https://github.com/clojure-emacs/clj-refactor.el`, [Online; accessed 27-May-2018].

Cunningham, W. (1992). The wycash portfolio management system. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, New York, NY, USA: ACM, OOPSLA '92, pp. 29–30.

Erné, M. et al. (1993). A primer on galois connections. *Annals of the New York Academy of Sciences*, 704(1), pp. 103–125.

Erwig, M. and Ren, D. (2004). Monadification of functional programs. *Science of Computer Programming*, 52(1), pp. 101–129.

Ferrari, D. and Lau, E. (1976). An experiment in program restructuring for performance enhancement. In *Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, pp. 146–150.

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

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9(8), pp. 28–35.

Free Software Foundation (2015). GNU Emacs. `https://www.gnu.org/software/emacs/emacs.html`, [Online; accessed 27-May-2018].

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].

GHC (2016). Welcome to Hackage! `https://hackage.haskell.org/`, [Online; accessed 21-November-2016].

GHC API (2016). ghc-7.10.3: The GHC API. `https://downloads.haskell.org/~ghc/7.10.3/docs/html/libraries/ghc/`, accessed : 2017-08-21.

GHC Developer Wiki (2017). Implementation of Trees that Grow. `https://ghc.haskell.org/trac/ghc/wiki/ImplementingTreesThatGrow`, [Online; accessed 28-August-2017].

Griswold, W. G. (1992). *Program Restructuring As an Aid to Software Maintenance*. Ph.D. thesis, Seattle, WA, USA, uMI Order No. GAX92-03258.

Gröber, D. (2017). The ghc-mod package. `http://hackage.haskell.org/package/ghc-mod`, [Online; accessed 25-June-2017].

Hallgren, T. (2003a). Haskell tools from the Programatica project. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, ACM, pp. 103–106.

Hallgren, T. (2003b). Haskell tools from the programatica project. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, New York, NY, USA: ACM, Haskell '03, pp. 103–106.

Haskell-tools (2017a). Haskell Tools. `http://haskelltools.org/`, [Online; accessed 01-August-2017].

Haskell-tools (2017b). haskell-tools Github. `https://github.com/haskell-tools/`, [Online; accessed 01-August-2017].

HaskellWiki (2015a). Monad class. `http://wiki.haskell.org/Monad`, accessed : 2016-07-25.

HaskellWiki (2015b). Polymorphism. `https://wiki.haskell.org/Polymorphism`, accessed : 2018-07-22.

HaskellWiki (2015c). Programming guidelines. `https://wiki.haskell.org/Programming_guidelines#IO`, [Online; accessed 14-August-2017].

HaskellWiki (2017a). GHC/As a library. `https://wiki.haskell.org/GHC/As_a_library`, accessed : 2018-08-06.

HaskellWiki (2017b). Language extensions. `https://wiki.haskell.org/Language_extensions`, [Online; accessed 27-May-2017].

Hickey, R. (2018). The Clojure Programming Language. `https://clojure.org`, [Online; accessed 27-May-2018].

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146, pp. 29–60.

Horpácsi, D. (2013). Extending erlang by utilising refactorerl. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, ACM, pp. 63–72.

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

JetBrains (2016). PSI Files. `http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_files.html`, [Online; accessed 16-July-2018].

Jones, I. and Coutts, D. (2016). The Cabal package. `http://hackage.haskell.org/package/Cabal`, [Online; accessed 23-November-2016].

Koppel, J. and Alan, Z. (2018). Strafunski-StrategyLib. `https://github.com/jkoppel/Strafunski-StrategyLib`, [Online; accessed 16-June-2018].

Kort, J. and Lämmel, R. (2003). A framework for datatype transformation. *Electronic Notes in Theoretical Computer Science*, 82(3), pp. 463–482.

Kuck, D. J. (1981). Automatic program restructuring for high-speed computation. In *International Conference on Parallel Processing*, Springer, pp. 66–84.

Lämmel, R. (1999). Reuse by program transformation. In *Scottish Functional Programming Workshop*, pp. 144–153.

Lämmel, R. (2002). Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, New York, NY, USA: ACM, RULE '02, pp. 15–28.

Lämmel, R. and Jones, S. P. (2003). Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, New York, NY, USA: ACM, TLDI '03, pp. 26–37.

Lämmel, R. and Visser, J. (2002). Typed combinators for generic traversal. In *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, pp. 137–154.

Leather, S. et al. (2015). Type-changing rewriting and semantics-preserving transformation. *Science of Computer Programming*, 112, pp. 145–169.

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

Li, H. (2006). *Refactoring Haskell Programs*. Ph.D. thesis, University of Kent.

Li, H. and Thompson, S. (2011). A Domain-Specific Language for Scripting Refactoring In Erlang. Tech. rep., Technical Report 5-11, School of Computing, Univ.

Li, H. and Thompson, S. (2012). Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, ACM, pp. 32–39.

Li, H., Thompson, S. and Reinke, C. (2005). The Haskell Refactorer, HaRe, and Its API. *Electronic Notes in Theoretical Computer Science*, 141(4), pp. 29–34.

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

Li, H. et al. (2008). Refactoring with wrangler, updated: Data and process refactorings, and integration with eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, New York, NY, USA: ACM, ERLANG '08, pp. 61–72.

Lipovača, M. (2012). *Learn you a haskell for great good!: a beginner's guide*. no starch press.

Luposchainsky, D. (2014). Haskell 2014: 'Applicative => Monad' proposal (AMP). `https://github.com/quchen/articles/blob/master/applicative_monad.md`, accessed : 2014-05-21.

Mainland, G. (2007). Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, ACM, pp. 73–82.

Marlow, S., Newton, R. and Peyton Jones, S. (2011). A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, New York, NY, USA: ACM, Haskell '11, pp. 71–82.

Marlow, S. and Peyton Jones, S. (2012). The Architecture of Open Source Applications. Volume II: Structure, Scale, and a Few More Fearless Hacks. chap. The Glasgow Haskell Compiler.

Marlow, S. et al. (2010). Haskell 2010 language report. *Available online* `http://wwwhaskellorg/(May2011)`.

Marlow, S. et al. (2014). There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA: ACM, ICFP '14, pp. 325–337.

Marlow, S. et al. (2016). Desugaring Haskell's Do-notation into Applicative Operations. In *Proceedings of the 9th International Symposium on Haskell*, New York, NY, USA: ACM, Haskell 2016, pp. 92–104.

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].

Mens, T., Demeyer, S. and Janssens, D. (2002). Formalising behaviour preserving program transformations. In *ICGT*, vol. 2, Springer, pp. 286–301.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3), pp. 348–375.

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

Najd, S. and Jones, S. P. (2017). Trees that grow. *Journal of Universal Computer Science*, 23(1), pp. 42–62.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks*. Ph.D. thesis, Champaign, IL, USA, uMI Order No. GAX93-05645.

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

Philip, J.-F. (2012). GeneralizedNewtypeDeriving is Profoundly Unsafe. `http://joyoftypes.blogspot.com/2012/08/generalizednewtypederiving-is.html`, [Online; accessed 27-May-2018].

Reade, C. (1989). *Elements of functional programming*. Addison-Wesley.

Reinke, C. et al. (2005). Monadification as a refactoring. `http://www.cs.kent.ac.uk/projects/refactor-fp/Monadification.html`, accessed : 2014-05-19.

Renke, C. (2005). Type-Directed Monadification. `http://community.haskell.org/~claus/talks/`, accessed : 2014-05-19.

Röjemo, N. (1995). *Garbage collection, and memory efficiency, in lazy functional languages*. Chalmers University of Technology.

Rowe, R. N. S. and Thompson, S. J. (2017). Rotor: First steps towards a refactoring tool for ocaml. In *OCaml Users and Developers Workshop 2017*.

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].

Stack contributors (2017). The Haskell Tool Stack. `https://docs.haskellstack.org/en/stable/README/`, [Online; accessed 24-June-2017].

Stewart, D. and Leather, S. (2017). The dlist package. `https://hackage.haskell.org/package/dlist`, [Online; accessed 19-July-2017].

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, Error-Correcting Combinator Parsers. In J. Launchbury, E. Meijer and T. Sheard, eds., *Advanced Functional Programming*, *LNCS-Tutorial*, vol. 1129, Springer-Verlag, pp. 184–207.

SYB Package (2014). The syb package. `http://hackage.haskell.org/package/syb-0.4.1`, accessed: 2014-02-13.

Thompson, S. (2011). *Haskell: the craft of functional programming*, vol. 2. Addison-Wesley.

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

Thompson, S. et al. (2010). Refactoring Functional Programs. `http://www.cs.kent.ac.uk/projects/refactor-fp/`, accessed : 2014-05-13.

Tools, R. (2017). HaRe : The Haskell Refactorer. `https://github.com/RefactoringTools/HaRe`, [Online; accessed 16-June-2018].

Tóth, M. and Bozó, I. (2012). *Static Analysis of Complex Software Systems Implemented in Erlang*, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 440–498.

Visser, E. (2004). Program transformation with Stratego/XT. In *Domain-Specific Program Generation*, Springer, pp. 216–238.

Volkov, N. (2016). html-tokenizer: An "attoparsec"-based HTML tokenizer. `http://hackage.haskell.org/package/html-tokenizer`, [Online; accessed 25-November-2016].

Wadler, P. (1995). Monads for functional programming. In *International School on Advanced Functional Programming*, Springer, pp. 24–52.

Wadler, P. (1999). How enterprises use functional languages, and why they don't. In *The Logic Programming Paradigm*, Springer, pp. 209–227.

Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda-Calculus.* Ph.D. thesis, University of Oxford.

Weirich, S. (2016). Pi-Forall language. `https://github.com/sweirich/pi-forall`, [Online; accessed 25-November-2016].

Yorgey, B. (2009a). Abstraction, intuition, and the "monad tutorial fallacy?". `https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/`, [Online; accessed 27-May-2018].

Yorgey, B. (2009b). Typeclassopedia. `https://wiki.haskell.org/Typeclassopedia`, [Online; accessed 13-July-2017].

Zimmerman, A. (2015). This is a decription of the API Annotations introduced with GHC 7.10 RC2. `https://ghc.haskell.org/trac/ghc/wiki/ApiAnnotations`, [Online; accessed 04-April-2017].

Zimmerman, A. and Pickering, M. (2016). ghc-exactprint. `https://github.com/alanz/ghc-exactprint/`, [Online; accessed 23-November-2016].