TOOL SUPPORT FOR
REFACTORING HASKELL PROGRAMS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

By

Christopher Mark Brown

September 2008

Contents

List of Tables

List of Figures

Publication

I, Christopher Mark Brown, declare that the work in Chapter 3 has been previously published [13] in a conference proceedings and that I am the main author of the work.

Abstract

Refactoring is the process of changing the internal structure of a program, while preserving its behaviour. The behaviour preservation is crucial so that refactorings do not introduce any bugs. Refactoring is aimed at increasing code quality, programmer productivity and code reuse. Refactoring has been practised manually by programmers for as long as programs have been written; however, with the advent of refactoring tools, refactoring can be performed semi-automatically, which allows refactorings to be performed (and undone) easily.

This thesis reports research into a set of program refactorings for the functional programming language Haskell, using the framework of the Haskell Refactorer, HaRe. The program refactorings generally fall into two categories. Structural refactorings, that affect programs at the *expression* level, and type-based refactorings that affect programs at both the *type* level and the *expression* level. Type-based refactorings use the type checker as part of the refactoring process.

Furthermore, the thesis looks into deriving a program slicing technique for Haskell programs and defines a set of refactorings used to help eliminate dead code from a program. Haskell clone detection, that is, detection of duplicate code, and removal is also discussed.

A case study into refactoring Haskell programs is also given; the case study not only looks into using the refactorings described in this thesis, but also at general ways to refactor Haskell systems.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Simon Thompson, for his amazing intellect and guidance. I have never met anyone with such a high level of knowledge, dedication and encouragement. His guidance has always been invaluable and it has often been a useful reality check for getting me through the tough times.

I would like to thank Dr. Dave Harrison, for being the un-hired help, and also for being a great friend. Our meetings have always been useful, and without his support I probably would never have finished.

I thank my family for their amazing support and patience throughout my university life.

I thank the members of the Computing Lab at the University of Kent. In particular, my colleague Dr. Huiqing Li for her ongoing help and patience.

A special thanks goes to my friends Daniel Grundy, Thomas Davie and Toby Huitson for being there when the going got tough. I thank them for helping me get through this, all of them have helped much more than they will probably ever realise.

I would like to extend my thanks to George and Pauline Urwin for allowing me stay at their house in the final stages. Their hospitality has been very generous, it is not often we meet such generosity these days. A personal thanks to George, for allowing me to appreciate that there is more to life other than functional programming.

Chapter 1

Introduction

This thesis is concerned with the investigation and implementation of a number of refactorings for Haskell. The refactorings are themselves written in Haskell [108], and implemented using the framework of HaRe, the Haskell Refactorer [68]. This chapter is aimed to give an overview of this thesis. In particular, Refactoring in general is described in Section 1.1 (Page 1); Section 1.2 (Page 6) describes what it means to refactor functional languages, including giving a brief overview of Haskell in Section 1.2.1 (Page 6). We then given an overview of the refactorings outlined in this thesis in Section 1.3 (Page 13); the contributions of the thesis are given in Section 1.4 (Page 13); finally, a summary of the proceeding chapters is given in Section 1.5 (Page 16).

1.1 Refactoring

Often programmers write a first version of a program without paying full attention to programming style or design principles [54]. Often, having written a program, the programmer will realise that a different approach would have been much better, or that the context of the problem has changed. Refactoring tools provide software support for modifying a program into a better written program thus avoiding the expense of re-starting from scratch.

Refactoring is the process of changing the internal structure of a program, while preserving its behaviour. The term refactoring was first introduced by Opdyke in his PhD thesis in 1992 [86] and the concept goes back to the fold/unfold system proposed by Darlington and Burstall in 1977 [14]. The key aspect of refactoring —in contrast to general program transformations, such as genetic programming [44] — is the focus on purely structural changes rather than changes in program functionality. Specifically, the advantages of refactoring are as follows:

- Refactoring is aimed at improving the design of software. Without refactoring, the design of a program will decay. As people change code —changes to realize short-term goals or changes made without a full comprehension of the design of the code—the code loses its structure. Regular refactoring helps to tidy up the code and to retain its structure.

- Refactoring makes software easier to understand. Programmers often write a program without thinking about future developers. They may understand their code at the moment that they are writing it, but in a short time the code may become completely incomprehensible. Refactoring helps the programmer to make their code more readable and therefore more amenable to change. A small time spent refactoring can make the code better communicate its purpose.

  Refactoring can also be used for a related purpose. Refactoring can be used to understand unfamiliar code, as programs are written in a very idiomatic style, refactoring can be used to change a program that is written in the style of one programmer so that it suits the style of another. Refactoring in this way helps to understand the design of the code.

- Refactoring encourages code reuse. Poorly designed code often has lots of repetition, i.e. identical code blocks that are the result of cut and paste routines. An important aspect of improving the design of a program is to eliminate duplicate code from the program. The importance of removing duplicate code lies in the ability to improve and maintain the code at a later stage. By removing the duplicate parts, the programmer is ensuring that the code says everything once and only once, which is one of the principles of good design.

- Refactoring helps the programmer to program faster. Refactoring encourages good program design, which allows for a development team to better understand their code. A good design is essential to maintaining speed in software development. Refactoring helps the programmer develop software more rapidly, because it stops the design of the system from decaying.

- Refactoring helps the programmer find bugs. As refactoring helps improve the design and understanding of a program, it helps the programmer to verify certain assumptions they've made about the program.

When a programmer learns a new technique that greatly improves their productivity, it is difficult to see when the technique does not apply. The following describes some of the pitfalls of refactoring:

- Changing Interfaces. *Renaming* a function, for example, changes the interface of a module. This can be potentially very dangerous if the renamed function was part of an API or a large development project. It could be that the other programmers of the system don't expect the renaming to have occurred.

- Refactoring and Design. Refactoring has a special role as a complement to design. Many people consider design to be the key piece and programming just the mechanics. One argument is that refactoring can be an alternative to upfront design, and could be relied upon by the programmer to implement the first approach that seems to work and then refactor it at a later stage. Refactoring is therefore not an excuse for avoiding serious thinking about program design.

The refactorings that we consider in this thesis are implemented in the Haskell Refactorer, HaRe. HaRe is the result of the combined effort of the *Refactoring Functional Programs* project at the University of Kent [71]. HaRe provides refactorings for the full Haskell 98 standard [93], and is integrated with the two most popular development environments for Haskell programs [95]: Vim [87] and (X)Emacs [15]. HaRe refactorings can be applied to both single- and multi- module programs; HaRe is itself implemented in Haskell, and is built upon the Programatica [40] compiler front-end, and the Strafunski [65] library for generic tree traversal. The HaRe programmers' application programmer interface (API) provides the user with an abstract syntax tree (AST) for the program together with utility functions (for example, tree traversal and tree transforming functions) to assist with the implementation of refactorings.

Pure functional programs are referentially transparent [105], therefore the opportunities for refactoring are much greater than for imperative programs. The classical example of this, of course, is that in a functional language it is always possible to transform `f x + g x` into `g x + f x` (assuming, of course, that `+` is a binary commutative operator). This is not possible for an imperative language because either `f` or `g` may change the value of the parameter `x` and therefore the result will depend on execution order, this could also happen if `x` or any global variable name is used in the computation of `f / g`. In addition to this, in a functional language duplicated occurrences of an expression (within the same scope) will have identical values, and so could be replaced with a shared computation; this is also not the case for imperative programs.

Refactoring can occur at any level within a program: some refactorings may occur within the scope of a function, say, and some may occur within the entire program scope. The *renaming* refactoring, for instance, renames all instances of a particular name within the scope of the entity where the name is declared; whereas, renaming a global name (such as a top-level function name, say) can potentially affect the entire program (as long as the refactoring can be performed at all).

A refactoring can either be atomic or composite. An atomic refactoring cannot be decomposed into simpler refactorings, whereas a composite refactoring can be decomposed into a series of smaller refactorings. For example, the *introduce* *pattern matches* refactoring, described in Section 5.6.1 could have been described in terms of the following atomic refactorings: *introduce sub-pattern*, *unfolding* and *case analysis simplification*.

Refactoring is actually a very common process and is usually practised implicitly, as programmers modify their existing code base. After writing a working program, a programmer may refactor the program to improve the programs design; they also may refactor the program to make the program more amenable to change, before changing the functionality of an existing system.

In the last decade a diverse range of refactoring tools have been developed for various programming languages. Before then, refactoring was typically done manually or with the help of text editor "search and replace" facilities. Manual refactoring is tedious, error-prone and, in most cases, slow. Recently developed refactoring tools help programmers refactor by offering a selection of automatic refactorings for their code base (a selection of refactoring tools is listed below; a more detailed review of refactoring tools is given in Chapter 3). Since the first successful refactoring tool: the *Refactoring Browser* [97], which supports Smalltalk refactoring, there has been a growing number of refactoring tools for a variety of different languages such as: Java, C, C++, C#, Python and UML [33]. The most recent release of the IntelliJ IDEA refactorer supports 35 Java refactorings [11]. Manual refactoring requires many extensive tests to ensure that the functionalities of the program are preserved [32]. While extensive tests can show up many bugs and improve an overall confidence, they cannot prove the correctness of a program.

When constructing a refactoring tool, there are two main activities to consider: *program analysis* and *program transformation*. Program analysis checks whether certain side-conditions (side-conditions are used in this thesis as an equivalent notion to pre-conditions), which are necessary for the refactoring, are met. Program analysis also collects information needed during the program transformation phase. For example, the introducing as-pattern refactoring —as implemented as part of this thesis— in a Haskell program requires the following:

- An Abstract Syntax Tree (the AST), which provides a complete structural representation of the program in question.
- The token stream, which includes information regarding the program's layout and comments.
- The particular part of the AST where the as-pattern is to be introduced (including the scopes that the as-pattern will effect).
- The pattern to which the as-pattern will be bound.
- The name which will be given to the newly declared as-pattern.

Program transformation is performing the actual structural code changes that make up a given refactoring. Both these steps are amenable to automation, as manifested by the rich collection of existing work in many areas of software engineering including compiler construction [8] and program slicing [119].

1.2 Refactoring Functional Languages

This section comprises of two parts. Section 1.2.1 introduces the Haskell programming language which is used as the language to refactor for this thesis. Section 1.2.2 discusses refactoring functional programming languages, in particular Haskell, and outlines any special issues that arise from refactoring Haskell programs.

1.2.1 The Haskell Programming Language

Haskell is a programming language that belongs to the *functional* paradigm. It differs to imperative languages such as C++ and Java in many ways. In imperative languages, programs are composed of statements which can change a global state when executed (such as writing to a file, or changing a variable). In a functional language, programs are composed by evaluating expressions and control the use of state and mutable data such as global variables in a helpful way. The most recent standard is Haskell 98, and is defined in *Haskell 98 Language and Libraries: the Revised Report* [92]. Haskell is the result of many years of research into lazy functional languages. The purpose of this section is to give a brief over-view of Haskell, it is not intended to give a complete description of the language. A complete specification can be found in the Haskell 98 report. This section is aimed at giving a brief overview of some of the basic features of Haskell:

- **Referential transparency** is the idea that expressions have values, and that value will always be the same, within the particular scope or context it is evaluated in. Referential transparency is what makes a functional language pure by making variables immutable. By declaring `x = 6` we are really declaring an equation to state that `x` is bound to the value `6`. No matter where `x` is evaluated it will always give the result `6` provided it is used in the scope that it is bound.

- **Lazy evaluation** means that expressions and function arguments are only evaluated when they are really needed and duplicate expressions in the same scope can be replaced with a shared computation. To demonstrate this, we first consider a function written in C, which is evaluated using the strict, *call by value* semantics of C. Unlike Haskell which uses non-strict, *call by need* (or lazy evaluation) semantics, C evaluates its function arguments before the function body, regardless of whether they are needed or not. Consider:

  ```
  int f (int x, int y)
  {
    return x+x;
  }
  ```

  This function always diverges if we pass `(3+3)` for `x` and `(1/0)` for `y` as the arguments to `f` will be evaluated before the function call.

  In Haskell however, arguments are only evaluated once and when they are needed. The same function in Haskell would be as follows:

```
   f x y = x + x
```

With lazy evaluation, we can evaluate `f` by introducing a shared computation for the argument `x` as follows:

```
   f x y = k + k where k = x


   f (3+3) (1/0)
=> k + k where k = 3 + 3
=> k + k where k = 6
=> 6 + 6
=> 12
```

As the expression `(3+3)` is used twice on the right hand side of `f` it is introduced as a shared definition within a `where` clause. The expression `(1/0)` is never used, so Haskell never attempts to evaluate it.

- **Recursion** allows for iteration without mutable variables. To illustrate this, consider the following function that calculates the length of a list:

  ```
  length :: [a] -> Int
  length [] = 0
  length (x:xs) = 1 + length xs
  ```

  The length of an empty list (`[]`) is `0`, while the length of a non-empty list (where `x` is the head and `xs` is the tail of the list) is 1 plus the length of the tail of the list.

- **Higher-order functions** are functions that take other functions as their arguments or return functions as their results. In most cases, it is the latter that is more important, as a program can create functions dynamically. It is possible to have functions as arguments in Pascal or C, say, but this doesn't make them higher-order. For example, in Haskell it is possible to write a function to add `2` to every element of a list:

```
subtract2 [] = []
subtract2 (x:xs) = x - 2 : subtract2 xs
```

In the example we have two equations, defined with pattern matching. The first equation states that if we pass the empty list (`[]`) to `subtract2` we get `[]` back. The second equation states that if we pass a non-empty list to `subtract2` where `x` is the head of the list and `xs` is the tail, we can subtract `2` from the first element, and then subtract `2` from the rest of the list using recursion. It is possible to generalise this function further, by introducing a new argument for `subtract2` that allows us to apply any function to the elements of the list:

```
subtract2 f [] = []
subtract2 f (x:xs) = f x : subtract2 f xs
```

We can then call `subtract2` with `(-2)` as its first argument (the function `(-2)` is given to us by the partial application of the operator `(-)`). The definition of `subtract2` is now polymorphic: it can take a function that takes an argument of any type, and the implementation of `subtract2` is completely transparent to this.

- **Monads** provide one way of introducing side effects to a pure language. While the language remains pure and referentially transparent, monads can provide implicit state by threading it inside them. Allowing side effects only through monads and keeping the language pure makes it possible to have lazy evaluation that does not conflict with the effects of impure code. Even though the expressions are evaluated lazily, some parts of them are forced by monads to be evaluated in a specific order and the effects are properly sequenced. The following shows the example of a monad in Haskell, using the `do` notation (taken from Hutton's parser combinator library [48]):

```
p :: Parser (Char,Char)
```

```
p = do
       c <- item;
       item;
       d <- item;
       return (c,d)
```

In the above example, we define a parser that consumes three characters, throws away the second character, and returns the other two as a pair.

- **Data types** allow Haskell programs to be made up of meaningful and flexible representations of the data we wish to manipulate. To illustrate this, we show the Haskell built-in data type called `Maybe`. This data type can be used in only two possible ways, either `Nothing` or `Just` a value of any type:

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

  It is also possible to define functions over the data types we define by pattern matching. Pattern matching allows the program to direct the flow of evaluation. Consider the following function, which is also a Haskell built-in function:

  ```
  isJust :: Maybe a -> Bool
  isJust Nothing = False
  isJust (Just x)  = True
  ```

  Passing `Nothing` to `inJust` forces the first equation to be evaluated, returning the value `False`; passing `Just 42`, say, instead forces the second equation to be evaluated, returning the value `True`.

This research examines the application of a number of transformation and analysis techniques for Haskell programs, and also uses Haskell as the implementation language. Implementation is based upon the infrastructure of HaRe: the Haskell Refactorer [71], developed at the University of Kent by Li, Reinke and Thompson.

Using Haskell as the implementation language allows us to explore the usability of Haskell for implementing transformation and analysis tools and also allows us to find out how these transformations and analyses can aid in the development and maintenance of a large software system. To show an example of the *renaming* refactoring that is already implemented for HaRe, we rename the function `subtract2` from the previous example to `map` to better reflect its generalisation:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

It is important to note that this renaming will also rename all previous references to `subtract2` to `map`.

It is assumed that the reader is familiar with the Haskell 98 programming language for the remainder of this thesis.

1.2.2 The Refactoring Functional Programs Project

The *Refactoring Functional Programs* was a project carried out at the Computing Laboratory at the University of Kent. The aim of the project was to investigate refactoring functional programming languages, namely Haskell and Erlang. Two refactoring tools for quite different functional languages have emerged from the project: the Haskell Refactorer, HaRe, and the Erlang Wrangler, which are both results of the combined development work of Li, Reinke and Thompson [61, 71].

Functional programming languages differ from their imperative counterparts in both theory and practice. Some functional refactorings, such as renaming an identifier, removing an unused parameter, etc., are also valid for imperative languages. There are many refactorings, however, that are unique to functional programming. Adding a constructor to a data type, introduction/elimination of as-patterns and monadifying a particular expression (i.e. wrapping an expression of type `a` in `m a`), are all examples of functional programming specific refactorings. Developing a refactoring tool for a functional language was therefore not simply a re-implementation of an existing refactoring tool for an imperative language.

The layout of Haskell programs tends to be very individual and idiomatic — despite the fact that the offside rule [66] limits this to some degree— especially if a standard layout is not enforced by a particular programming editor. It is important, therefore, that the refactoring tool preserves as much of the programmer's layout style as possible. These aesthetics are important so that the programmer can recognize the program after the refactoring process; HaRe supports this reservation, and also retains the programmer's comments. It is important to note, however, that the programmer's comments may not reflect the program after the transformation has been made. A definition with explanatory comments, say, may result in a definition that appears meaningless if the definition is removed and the comments are still left.

The refactorings already supported by HaRe (and hence not a part of this thesis) fall into three categories: *structural* refactorings affecting names, scopes and the structure of the entities in question; *module-aware* refactorings affecting the imports and exports of the program; and *data-oriented* refactorings of data types.

Continuing the refactoring tool support for functional languages, the University of Kent also developed the Erlang Wrangler. Like HaRe, the Wrangler is aimed at the working Erlang programmer and is integrated into the (X)Emacs editor, and some preliminary work has been done on integrating Wrangler with Eclipse [28]. Unlike Haskell, Erlang is a strict, dynamically typed functional-based programming language with support for functions, concurrency, communication, distribution and fault-tolerance [4]. In contrast to Haskell —which arose from an academic initiative— Erlang was developed by the Ericsson Computer Science Laboratory, and like Haskell, has been actively used in industry both within Ericsson and beyond. Hitherto, the Erlang Wrangler has support for a number of refactorings, including: *rename an identifier*, *generalize a function definition* and *clone extraction*; the project is still a work in progress.

The aim of the research reported in this thesis is to build upon the existing work of Li [67], using HaRe as an infrastructure, to investigate the development of further (and in some cases more complex) refactorings, transformations and analyses for Haskell programs.

1.3 Refactoring Outline

The structure of the transformations that form a part of this thesis will proceed as follows:

- Data type based. Refactorings in this area include: *add a constructor to a data type*, *remove a constructor from a data type*, *add a field to a data type*, *remove a field from a data type* and *introduce pattern matching.*

- Structural based. Refactorings in this area include: *folding*, *generative folding*, *folding as patterns*, *unfolding as patterns*, *converting between let and where*, *converting between where and let* and *case analysis simplification.*

- Miscellaneous. These fall into two categories: program slicing and duplicate code elimination.

  – Program Slicing. Transformations within this category include: *dead code elimination*, *splitting* tuple returning functions, *merging* tuple returning functions and *an argument instantiation algorithm.*

  – Duplicate Code elimination defined as *abstraction.*

Figure 1 shows a diagrammatical overview of the refactoring structure.

1.4 Contributions of this Research

The study of this thesis was carried out as part of the *Refactoring Functional Programs* project. This study focuses on adding additional refactorings, transformations and analyses for the Haskell refactoring tool. The following describes a brief summary of the contributions; a complete overview of the contributions, in more detail, is discussed in Chapter 9, on Page 216.

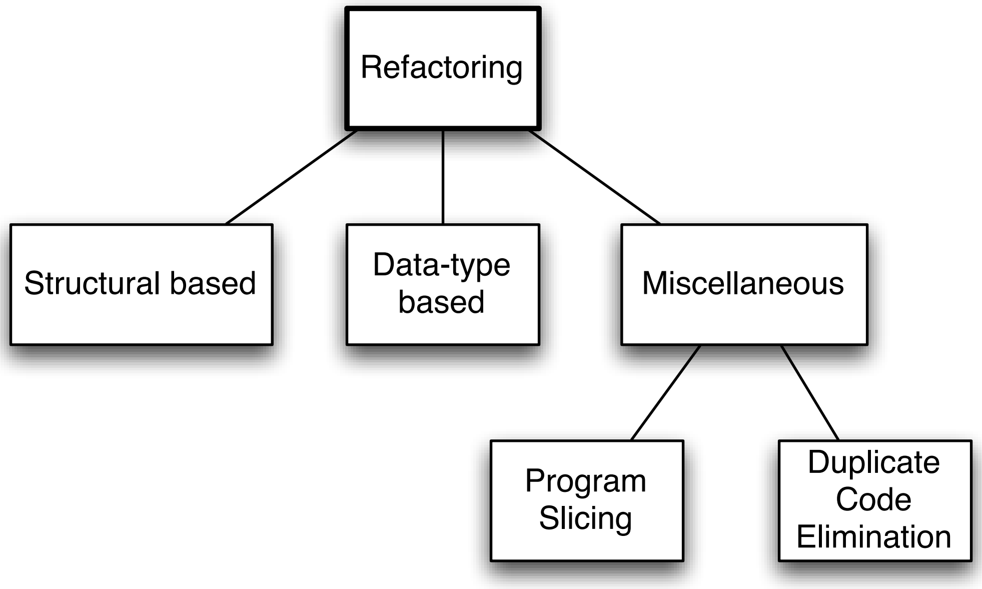


Figure 1: The structure of the refactorings

- Extending the HaRe framework and API for the creation of more complex refactorings. Previously, the implementation of HaRe had support for a number of stuctural refactorings. The HaRe framework and API, however, has been extended to be able to cope with the implementation of larger, more complex refactorings as part of this thesis. In particular, the API and framework has been extended to be able to introduce type-aware refactorings. This required that HaRe had access to —and was able to transform and traverse— type information for the Haskell program under refactoring.

- The design and implementation of a large set of structural refactorings. We have extended and complemented the already existing set of refactorings by adding additional structural refactorings. The refactorings are, in general, more complex and require a more developed refactoring framework.

- The design and implementation of a set of type-aware refactorings, in particular refactorings that transform programs at the type level: directly affecting the data types of the program; and refactorings that transform programs at a structural level: directly affecting the terms under a specific type constraint. The aim of the implementation of type-aware refactorings was to extend the HaRe framework to firstly deal with Haskell type checking —particularly in the context of Haskell program transformation and analysis, and secondly to complement and extend HaRe's catalogue of structural refactorings.

- Defining program slicing in a new way for functional programming and deriving a set of program slicing based refactorings. Currently, there has been little work in researching program slicing techniques for Haskell. We therefore have defined a new program slicing technique for Haskell programs called splitting; merging, the converse of splitting, is also defined. There are also a number of other smaller refactorings that were implemented as a result of splitting and merging.

- Clone detection (i.e. the discovery of duplicated code) and removal in Haskell. We present a clone detection technique to find duplicated code within Haskell programs and we also provide suitable transformations and refactorings to eliminate the duplicated code found. We apply clone detection and removal to a short example, and show how eliminating duplicate code can greatly increase the maintainabilty and understanding of a Haskell program.

- A case study for refactoring Haskell programs. In particular we apply the refactorings implemented in this thesis to a large-scale project. In performing the case study, we investigate the usefulness of the refactorings and the scope to which they can be applied. The case study also demonstrates the capacity of the HaRe tool in applying refactoring to large-scale Haskell programs. The intention of the case study is to also give some ideas for refactoring in the scope of future work. Finally an expression processing example is used to demonstrate the capacity of the refactorings from this thesis in a simple, but still useful, context.

- Reflecting on the suitability of Haskell as a language for building refactoring, transformation and analysis tools for the programming languages. In implementing a large set of these refactorings for Haskell programs, a lot of experience is gained by building tools that transform and analyse Haskell programs. In particular, experience is gained in using available technology, including compiler front-ends and traversal libraries. By implementing such a large number of complex refactorings we effectively evaluate the use of Haskell as a large scale programming language and the current state of the art language tools available for it.

1.5 Thesis Outline

The thesis will proceed as follows, where the conclusions are given in the final chapter:

Chapter 2: Tools for refactoring

An overview of the infrastructure of HaRe is given in this chapter. In particular, an overview of the Programatica [1] toolset and Strafunski [64] are presented together with the advantages and disadvantages of using them. The chapter briefly discusses other tools that could have been used for HaRe, and explains the reasoning behind the current rationale.

A working example of the HaRe tool in practice to refactor a simple program is given, together with an explanation of how the to-as-patterns refactoring is implemented in complete detail, using real code from HaRe.

Chapter 3: Program Slicing

A new program slicing technique for a functional language is discussed in this chapter. In particular, the chapter introduces a new program slicing technique for Haskell called *splitting*. The converse of splitting, namely *merging* is also defined, together with constituent smaller refactorings that make up the slicing implementation.

Chapter 4: Structural Refactorings

This chapter contains a set of new structural refactorings for HaRe; their description and conditions are given. Structural refactorings affect the expression level of a program, including function definitions. The chapter concludes by describing a technique known as symbolic evaluation, which is used to aid the transformation process in the case analysis simplification process.

Chapter 5: Data Type based refactorings

A description of the implementation of a set of new type-based refactorings for HaRe is given. Type-based refactorings are refactorings that affect the type definitions of a program, or affect the expressions of the program taking into account a type constraint.

The refactorings described in this chapter are as follows: adding and removing a constructor, adding a removing a data type field, introducing pattern matching and introducing a case expression.

Chapter 6: Clone Detection and Removal

This chapter presents a case study of duplicate code elimination within Haskell. A collection of new transformations and analyses for HaRe is presented that focus on the detection and elimination of duplicate code within Haskell.

In particular, the clone elimination is described in two parts: the first part is an automatic analysis system that searches for clones within a Haskell project and displays them in a report; the second part is a transformation process which allows the user to step through the clones identified by the analysis stage and replace them with a call to an abstraction. An example of how the clone detection and elimination routine works on a small example is also shown.

Chapter 7: A Refactoring Case Study

This chapter presents a case study of refactoring within a large Haskell project. Specifically, we look at refactoring a mainstream Haskell compiler, and use the example to investigate possible future refactorings. Finally, the chapter concludes with an extended example of refactoring a simple expression processing example.

Chapter 8: Related Work

This chapter presents the related work relevant to refactoring. Specifically, the chapter discusses work related to refactoring tools (which includes refactorings tools for languages that are not necessarily functional) and the generality of refactoring in functional languages. The chapter concludes with a section discussing current support for refactoring tools; this may include refactoring meta-languages and techniques to help the programmer in designing and applying refactorings.

Chapter 9: Conclusions and Future Work

This chapter gives a reflection on implementing refactorings for Haskell and the broader application to language design and analysis. A refection on the thesis as a whole, the approach taken, what was learned about refactoring and the successes and failures of the attempted work are presented. The chapter discusses the contributions of the thesis and any steps for future work.

Chapter 2

Tools for refactoring

This chapter describes the infrastructure on which HaRe is built, and also discusses the experience of the author and of the Kent group when designing and implementing HaRe and its refactorings. Section 2.2 discusses the Programatica toolset; Section 2.3 discusses the Strafunski library for generic tree traversal; in Section 2.4 the GHC compiler system is discussed; the HaRe system is described in Section 2.5; the chapter concludes with a description of a refactoring from HaRe in Section 2.7.

2.1 Introduction

As previously mentioned in Chapter 1, the Haskell Refactorer (HaRe) was originally designed and implemented by the *Refactoring Functional Programs* team (the Kent Group) at the University of Kent. The Kent group comprised of Simon Thompson, Huiqing Li and Claus Reinke [68].

The goal of the Kent group was to implement tool support for refactoring Haskell programs. Like text editors, refactoring tools support interactive program manipulation; however, refactoring tools need to query and modify the program syntax and semantics instead of the program as a character string. Rather than writing a user interface from scratch, it was decided that the refactoring tool was to be implemented with support from the most popular program development environments for Haskell users. Such a design decision allows the refactoring tool to be more accessible, and also means that programmers do not have to learn a new development environment in order to be able to refactor their code. There is an obvious disadvantage to reusing user interfaces however, namely that most interfaces do not offer all the facilities that the refactoring tools require; this has affected the design of the refactoring tool.

In order to manipulate a program syntactically, a refactoring tool must be able to retrieve and modify information regarding program declarations, identifiers, expressions and types. This information is stored in an AST (Abstract Syntax Tree) – typically generated by a parser. ASTs offer a reasonable solution for program modification on a syntactical basis: they provide a structural representation of the program using a collection of recursive data types, omitting unnecessary syntactic constructs such as brackets and spaces. An example of the Haskell data type representation for the AST used for HaRe is given in Figure 2 on Page 26.

In addition to syntactic information, a refactoring tool also requires the use of some static semantic information. Static semantic information includes scope information, type information and module information of the program under scrutiny. Scope information consists of the name space of all the identifiers in a program and binding information for where the identifiers are bound. Type information consists of the type and kind information of identifiers within the program. Module information records the interfaces of individual modules contained within the program and the module's inclusions. This information is obtained via static analysis, type analysis and module analysis.

In order for HaRe to retain comments and layout of Haskell programs, this information could ideally be stored in the AST; the pretty-printer (which produces program sources from ASTs) could make use of it during the pretty-printing process. Unfortunately at the time of the development of HaRe, parser front ends did not record this information in their ASTs. Therefore, effort from the Kent Group was required to modify the front end in order to record comments and layout information while still utilizing one of the existing Haskell front ends.

At the time of designing HaRe (2002) the Kent group compared and examined the following different Haskell compiler front-ends: GHC [77], Haddock [76], Hatchet [75] and Programatica [1]. The Kent group decided to use Programatica: it was the only compiler front-end available at the time that had support for full Haskell 98, together with more complete AST information than any of the other systems available.

Investigation was also made into generic programming libraries to support transformations over large abstract syntax trees. Although, in theory, program analysis and modification can be written for large mutually recursive data types without support for generic programming, the result is always large and cumbersome. While Haskell supports some general higher order functions such as `map` and `fold`, it is still not convenient to program over large, complex, recursive nested data type representations of Haskell. Instead, a generic programming technique to allow high-level program analysis and modification was needed; this generic programming technique was Strafunski [64, 65], although others were considered. At the time the project was started, Strafunski had just stabilised. Strafunski is a Haskell-based software bundle developed for supporting generic programming in application areas that involve term traversals over large abstract syntax trees. If HaRe were to be built now, it would probably use the GHC-API [77] and perhaps Scrap Your Boilerplate [53], for generic tree traversal.

A survey [95] was conducted by the Kent group in July 2002 to find the two most widely used development environments for Haskell. As a consequence, HaRe is currently integrated into (X)Emacs [15] and Vim [87]. HaRe also covers the full Haskell 98 standard language. The architecture of HaRe has been extended to include an API [71], exposing HaRe's infrastructure for implementing refactorings or general purpose program transformations. The API is intended for programmers who wish to write their own refactorings, rather than people who simply want to use HaRe.

2.2 The Programatica Toolset

Programatica [1] is a Haskell project developed at the OGI School of Science & Engineering, Oregon Health & Science University and subsequently at PSU (Portland State University). Programatica was implemented in Haskell as an interactive environment for the development of high-assurance Haskell software. The development environment that Programatica provides allows the program and its properties to be simultaneously developed and improved. To this purpose, the Programatica team have developed an expressive logic, called P-logic, and they have extended Haskell to support property definitions and assertions in P-logic. Source code that is written in Programatica's extended Haskell can include both definitions of executable code and assertions of properties; an assertion of properties can be accompanied by a certificate which encapsulates the evidence for this assertion.

The components of Programatica's front-end include a lexer, parser, type checker, module analysis system and also a pretty printer. Programatica supports the full Haskell 98 standard together with a number of Haskell extensions. HaRe uses all the parts of the Programatica front end in turn, apart from the type checker, which proved too slow for the development of the refactorings in this thesis. A brief description of each of these components is given in the following parts of this section. We discuss these details because we'll need to refer to aspects of the Programatica system in describing how our refactorings are implemented.

2.2.1 The Lexer

The implementation of the Programatica Lexer was generated by a regular expression compiler. The regular expression compiler reads the Haskell 98 grammar from the report and generates the main part of the Lexer, namely, the token recognition system, automatically. The type of the Lexer is defined as follows:

```
type Lexer = String -> [(Token, (Pos, String))]
```

`String` represents the Haskell program source. `Token` represents a classification of different types of tokens (defined as a data type) i.e. whether a token is an identifier or a keyword, for example; `Pos` represents the token's position (in terms of row and column numbers); the final `String` represents the content of the token.

Previous Haskell Lexers typically computed token positions and discarded whitespace while Programatica preserves this information in the token stream.

2.2.2 The Parser

The parser within Programatica is based on HsParser. HsParser is found in the haskell-src package of the Haskell hierarchical libraries. Programatica, however, uses its own lexer and the parameterised abstract syntax tree (see section 2.2.3 for details) for HsParser rather than the supplied lexer and abstract syntax tree that comes with HsParser.

The parser produces a variant of the abstract syntax tree in which every identifier is paired with its actual source location in the file. The source location is represented by a combination of the file name, and the position of its first character by the row and column number. The type of the source location for an identifier is `SN HsName`. `SN` and `HsName` are defined as follows:

```
data SN i = SN i SrcLoc
data SrcLoc = SrcLoc {srcPath :: FilePath,
                      srcChar, srcLine, srcColumn :: Int}
data HsName = Qual ModuleName String
            | UnQual Id
data ModuleName = PlainModule String
            | MainModule FilePath
```

A further analysis and modification of the AST adds additional static scope information to each identifier. The scoped AST is defined as follows:

```
type HsModuleP = HsModuleI ModuleName PNT [HsDeclI PNT]
```

The individual components are defined as follows:

- `[HsDeclI PNT]` is the list of identifiers occurring in the module; `PNT` is defined below.
- `ModuleName` is the name of the module, and whether or not the module is a main module or a plain module.

In the scoped AST, each identifier is associated with not only its actual source location, but also the location of its defining occurence and name space information. The type for these kind of identifiers is called `PNT` (Programatica Name Type) and is defined as follows:

```
data PNT = PNT (PN HsName Orig) (IdTy Pid) OptSrcLoc
```

The individual components are defined thus:

- `HsName` is the name of the identifier in question;
- `Orig` is the identifier's origin information (which usually contains the identifier's defining module and position);
- `IdTy Pid` specifies the category (i.e. variable, field name, type constructor, data constructor, class name, etc. of the identifier) with information of relevant type if the identifier is a field name, type constructor or data constructor.
- `OptSrcLoc` contains the identifier's source location information.

The complete definition of `PNT` and its component data types are given in Appendix B. Compared with normal ASTs, the scoped AST makes work easier for the tool writer in several respects:

- Source position information in the AST makes it easier to map fragments of code within the source file to their corresponding representation in the scoped AST.

- Identifiers can be distinguished looking at their `PNT` representations. Two identifiers are semantically equivalent if and only if they have the same origin therefore.
- Given an identifier, the scoped AST makes it easy to find the identifier's definition and use sites.

The program's layout can be calculated to some degree by using the source location within the scoped AST. In order to calculate the complete layout for a program, comments, keywords and some special characters, together with their location information, must also be represented in the scoped AST. This information is as yet not recorded in the scoped Programatica AST. Therefore, HaRe retrieves this information from the token stream supplied by the lexer.

2.2.3 The Abstract Syntax Tree

Programatica represents Haskell abstract syntax tree [93] by a parameterised syntax. The definition of a parameterised syntax has two levels: a structure defining level, and a recursive resolution level [103]. Figure 2 shows an example of the parameterised syntax for an expression, where:

- `i` represents the data type of the identifiers;
- `e` represents the data type of expressions;
- `p` represents the data type of patterns;
- `ds` represents the data type of declarations;
- `t` represents the data type of types;
- `c` represents the data type of a type context.

The type representing a module is:

```
data EI i e p ds t c
    = HsId (HsIdentI i)
    | HsLit SrcLoc HsLiteral
    | HsInfixApp e (HsIdentI i) e
    | HsApp e e
    | HsNegApp SrcLoc e
    | HsLambda [p] e
    | HsLet ds e
    | HsIf e e e
    | HsCase e [HsAlt e p ds]
    | HsDo (HsStmt e p ds)
    | HsTuple [e]
    | HsList [e]
    | HsParen e
    | HsLeftSection e (HsIdentI i)
    | HsRightSection (HsIdentI i) e
    | HsRecConstr SrcLoc i (HsFieldsI i e)
    | HsRecUpdate SrcLoc e (HsFieldsI i e)
    | HsEnumFrom e
    | HsEnumFromTo e e
    | HsEnumFromThen e e
    | HsEnumFromThenTo e  e e
    | HsListComp (HsStmt e p ds)
    | HsExpTypeSig SrcLoc e c t
    | HsAsPat i e
    | HsWildCard
    | HsIrrPat e
      deriving (Read, Show, Data, Typeable)
```

Figure 2: The parameterised syntax for expressions

```
data HsModuleI m i ds
    = HsModule { hsModSrcLoc  :: SrcLoc,
                 hsModName    :: m,
                 hsModExports :: Maybe [HsExportSpecI m i],
                 hsModImports :: [HsImportDeclI m i],
                 hsModDecls   :: ds }
      deriving (Eq, Show, Data, Typeable)
```

where:

- `m` represents the data type of the module name (whether or not the module is the main module or not);
- `i` represents the data type of identifiers;
- `ds` represents the data type of the declaration list.

Instantiating these parameters differently will result in different abstract syntax trees, which is useful as it allows one data type to be reused when representing different Haskell programs. The disadvantage to this two-level approach, however, is that it introduces an extra layer of tagging in the data structures.

The Haskell 98 syntax defined in Programatica contains 20 data types and 110 constructors in total. The data type defining expressions (as shown in Figure 2) contains 26 constructors. Writing AST traversals on this non-trivial mutually recursive abstract syntax tree without support for generic programming would produce large amounts of unwanted, and unreliable, boilerplate code. This would also negatively impact the maintenance and reusability of the produced code as boilerplate code is difficult to modify and understand.

2.2.4 The Module System

A formal specification of the Haskell 98 module system has been developed as part of the Programatica project by Diatchki, *et. al.* [25]. The specification is entirely written in Haskell. The semantics of a Haskell program with respect to the module system is a mapping from a collection of modules to their corresponding in-scope relation (the entities that are in-scope within the current module) and export relation (the entities that are exported by the current module). Given a list of modules, the analysis program reports either a list of errors found in each module, or returns the in-scope and export relations of the modules in question.

2.2.5 The Pretty Printer

The pretty printer within Programatica is based on the Pretty Printer Combinators of Hughes and Peyton Jones [46, 91] . Additional instances have been defined for the `Printable` class for each data type within the scoped AST for Programatica. The pretty printer does not use the source location stored within the AST as it does not need the location information to print a default layout for Haskell entities.

2.2.6 The Advantages and Disadvantages of Using Programatica

Programatica has allowed us to write large-scale refactorings over the full Haskell 98 standard. Programatica has a relatively straight forward front end which allows us to access the token stream and AST directly. Programatica also preserves comments and whitespace within the token stream. Furthermore, the entity relations between different Haskell modules are easily accessed from within Programatica, this allowed for the HaRe refactoring to be easily extended to work over multiple-module projects.

Unfortunately, after using Programatica as the front end for the HaRe system, a number of limitations were found:

- Programatica only supports Haskell 98. Most non-trivial Haskell programs now use the various extensions to the Haskell 98 standard provided with GHC. These extensions are not supported by Programatica. This has reduced the usefulness of HaRe for practising programmers.

- The type checking facilities in Programatica are very slow. Some of the refactorings implemented as part of this thesis require use of the type checking facilities. Obtaining such type information from Programatica has become a performance bottleneck. Therefore the type-checking facilities from GHC [77] were employed instead.
- Programatica is not distributed with any other compiler. HaRe must include the whole of the Programatica package as part of the general release. This, however, is a relatively minor issue.
- Some programming constructs are represented in the AST without location information, due to the fact that the Programatica project did not require this information. Examples of these constructs are parentheses and the empty list. This affects the layout of the refactored program as HaRe cannot correctly infer where in the program the constructs must be placed on pretty printing.

2.3 Strafunski - Generic Tree Traversal

Strafunski [64, 65] is a Haskell-based software bundle for implementing language processing components — most notably program analyses and transformations over large abstract syntax trees. The main idea behind Strafunski is to view traversals as generic functions that can traverse into terms while mixing uniform and type-specific behavior. Strafunski is composed of a number of functional strategies. A functional strategy is a function that can be applied to arguments of *any* type (as long as that type derives from `Typeable` and `Data`), can exhibit *type-specific* behavior, and can perform generic *traversal* into subterms.

Functional strategies are composed and updated in *combinator* style, that is, functional strategies are generic functions. The advantage of using Strafunski to write tree traversals and analyses as opposed to using standard techniques such as boilerplate programming is that it allows one to write concise, type-safe, generic functions for AST traversals, in which only the strictly relevant constructors need to be mentioned. Using Strafunski usually results in significantly shorter programs than the boilerplate written in plain Haskell. An earlier prototype implementation of the renaming refactoring "by hand" resulted in around 90% of code identified as boilerplate; this meant that the code was difficult and time consuming to write in the first place, and also meant that the code was very difficult to maintain.

In order to use Strafunski over an arbitrary nested data type, the data type must have class instances defined for `Typeable` and `Data`. `Typeable` and `Data` are Haskell type classes that contain members for type-safe cast and processing constructor applications [62]. The instances of these types can either be derived by a tool called DriFT [26], or they can be derived automatically by GHC. The latter requires a clause to derive `Typeable` and `Data` to be added to every relevant data type. The original approach in HaRe was to use DrIFT to derive the instances automatically; the instances are now derived automatically using GHC.

2.3.1 Strategy Types and Application

There are two kinds of strategies in Strafunski. The first kind, *TP*, models type-preserving strategies where the result of the strategy application for a term of type t is of type $m\ t$ (t in a monadic form). The second kind, *TU*, models type-unifying strategies where the result of the strategy application is always a monadic type $m\ a$ regardless of the type of the input term. For both cases, the result type is monadic in order to deal with effects in strategies such as state passing or non-determinism.

A type-preserving strategy is usually used for program transformation over a given layered data type, and type-unifying strategies are used for program analysis. *TP* and *TU* are represented as abstract data types and their exact definitions rely on the underlying models (for reasons of space the underlying models are not given in this thesis, but more details can be found in [64]).

Strafunski supplies a number of generic programming themes. The *Fixpoint-Theme* deals with iterative term transformation, stopping when a fixpoint is found. The *TraversalTheme* is used to define numerous traversal strategies. The *NameTheme* provides an abstract strategy for different kinds of name analysis.

The most common strategy theme provided by Strafunski is the recursive traversal theme. The recursive traversal theme provides the most heavily used strategies in the HaRe tool. There are four kinds of traversal strategy:

- Full Traversals. Apply a strategy to all nodes in a subtree.
- Single Hit Traversal. Process one subtree, terminating when the strategy succeeds.
- Stop Traversal. Descend into subtrees of failing nodes, proceeding into sibling nodes upon success.
- Traversals with Environment Passing. Start a traversal with a given environment, and modify the environment during the traversal process.

An overview of some of the combinators from the traversal theme that were used for the implementation of the refactorings for this thesis are given in Appendix A.1.

2.3.2 Integrating Strafunski with Programatica

The Strafunski generic tree traversal library is used by the refactorings implemented in HaRe to write generic program analysis/transformation functions over Programatica's abstract syntax tree. In this section two examples are given. The first example, taken from the *to-as-patterns* refactoring, shows a type-unifying strategy that collects some particular information about the abstract syntax tree, and the second example shows a type-preservation strategy that performs an abstract syntax tree modification.

The first example, shown in Figure 3, traverses a given term and returns the first occurrence of a pattern binding within that term, if that pattern is equivalent to a given expression. In the example, the functions `applyTU`, `once_tdTU` and `failTU` are all querying functions (type-unifying strategy combinators) from

```
getPat :: Term t => HsExpP -> t -> HsPatP
getPat exp t
 = fromMaybe (error "No Pattern is associated with expression!")
     (applyTU (once_tdTU (failTU `adhocTU` worker)) t)
       where
        worker (p :: HsPatP)
          | rewritePats p == exp = Just p
          | otherwise = Nothing

rewritePats :: HsPatP -> HsExpP
```

Figure 3: Collecting certain information about the abstract syntax tree

the *strategyLib* Haskell module that comes with the Strafunski package-bundle. `applyTU` applies a type-unifying strategy to a term, `t`. In the example, we have an abstract syntax tree, `t`, that we traverse to find all possible pattern bindings. We do this by applying a function, `worker` to every node of the tree. We stop traversing the tree as soon as `worker` can be applied to a node that is a pattern. This traversal behaviour is determined by `once_tdTU`. The following describes these functions in turn:

- `once_tdTU` traverses the tree in a top-down manner, terminating when the pattern match to `worker` succeeds.
- `failTU` is a polymorphic strategy that always fails (by using `mzero` from the `MonadPlus` class) regardless of the given term.
- `adhocTU` allows the function `worker` to be applied to all nodes in a layered data type: it updates a strategy to add type-specific behavior so that the function on the left can be applied unless the function on the right succeeds.

A single-hit traversal is used here rather than a full traversal as only the top most pattern binding in a given term is required. The function `worker` is applied to every pattern node in `t` suceeding when the pattern in question —after conversion to an expression (captured by `rewritePats`)— is equal to an expression passed in as a parameter.

```
rewriteExp :: (MonadPlus m) => String -> HsExpP -> HsExpP -> m HsExpP
rewriteExp name e1 e2
  = applyTP (full_tdTP (idTP `adhocTP` (inExp e1))) e2
     where
       inExp (Exp (HsParen e1)::HsExpP)
             (e2@(Exp (HsId (HsVar pnt@(PNT pname ty loc))))::HsExpP)
        = do
             if (rmLocs pnt) == (rmLocs name)
              then do
               return e1
              else do
                return e2
       inExp (e1::HsExpP)
             (e2@(Exp (HsId (HsVar pnt@(PNT pname ty loc))))::HsExpP)
         | findPNT pnt pname  = return e1
         | otherwise = return e2
       inExp e1 e2 = return e2
```

Figure 4: Performing an abstract syntax tree modification

The second example, shown in Figure 4, checks to see whether a given expression occurs within another expression. If it does, the expression is replaced with the name of an 'as' pattern. Using the combinators `applyTP`, `full_tdTP`, `idTP` and `adhocTP` from *StrategyLib*, the traversal performs a full top-down traversal over the abstract syntax tree for a given expression as defined by `full_tdTP`. Each node of the given abstract syntax tree is traversed, and the function `inExp e1` is applied whenever a node of type `HsExpP` is encountered. When the strategy traverses into nodes that are not of this type the polymorphic identity combinator, `idTP`, is applied. `inExp` looks for a name within an expression and checks to see if it is the same as `name` which is a parameter passed in referring to an as-pattern name. If it is the same then the first expression is returned, otherwise the expression node that is currently under scrutiny is returned instead.

2.3.3 Advantages and Disadvantages of Using Strafunski

Strafunski has many advantages in that it allows one to write robust and concise AST traversals which prevent lots of boilerplate code from being implemented.

However, the learning curve for Strafunski is particularly steep, especially for those who are not accustomed to functional programming and AST traversals in general. There are several aspects of Strafunski that may lead to confusion; what follows is a summary of the experience of the Kent Group when designing HaRe and its refactorings:

- Choosing the correct default strategy is crucial for different traversals. For instance, `idTP`, which returns the identity of the term under scrutiny, can be used as the default strategy for `full_tdTP` and `full_buTP`, whereas `failTP` cannot. For traversals that are type-preserving with stop conditions, such as `stop_tdTP`, `once_tdTP` and `once_buTP`, `failTP` (the always failing strategy) is the correct default strategy and `idTP` is not. This issue is discussed by Thompson and Lämmel in [63].

  For the type-unifying strategy `full_tdTU`, `constTU []` (the constant strategy that always returns the empty list) can be used. For type-unifying strategies with stop conditions, `failTU` is always the correct default strategy.

- It is not always clear when to use bottom-up or top-down traversals, and for most traversals, there is no difference. There is one special case, however: when a full traversal (such as `full_tdTP` or `full_buTP`) tries to extend a particular node of an AST with a larger node of the same type, `full_tdTP` tends to cause stack overflow errors, whereas `full_buTP` does not. This is because a `full_tdTP` will in general apply the argument strategy *prior to descent*. This may be problematic in case the argument strategy increases the depth of a given term [63].

- Two-layer nested monads appear when using type-unifying strategies. Monads are use to manage the backtracking behavior. The returned result should have two layers of monads:

  1. The first layer is for control flow, e.g. `Maybe` is the monad used by Strafunski for backtracking.
  2. The second layer is used for monadic data e.g. `List` monad is used for collecting data.

  Omitting one of these monads always gives unexpected results.

2.4 The Glasgow Haskell Compiler

The most commonly used Haskell compiler is the Glasgow Haskell Haskell Compiler, GHC [77]. GHC has become the *de facto* standard for Haskell. Initially GHC was designed as a compiler to allow researchers try out implementation ideas, but has since also become a production quality compiler that supports a large number of extensions to the Haskell 98 standard. Some of these extensions include: multi-parameter type classes, overlapping instances, arbitrary-rank polymorphism, un-boxed types, and syntactic extensions include: hierarchical modules, pattern guards, recursive do-notation and parallel list comprehensions.

There has been a long-standing demand for the GHC team to release an API for the GHC front-end, enabling practising programmers to build on the GHC infrastructure and produce their own tools (refactoring tools, editors, debuggers, etc.). Recognizing this, the GHC development team have provided an API that Haskell programmers can use to gain access to the internals of the compiler [77].

The GHC API exposes the entire internal infrastructure of GHC to Haskell programmers. The GHC API is mainly a large and sophisticated selection of functions and data types. The GHC team is currently in the process of developing a simplified version of the API intended for the use by utility writers. The main advantages to using the GHC API over the Programatica API are as follows:

- The GHC API provides access to all the GHC extensions, allowing tools to be integrated with the de facto Haskell standard.
- GHC is a well maintained Haskell compiler, and should therefore always support the latest language extensions.
- In the future the GHC API will provide a layer of insulation against the changes made to the internal organization of GHC.

An attempt was made by Ryder [102] to fully port HaRe across from Programatica to GHC. The work was done just as the API was first being defined; more recent work has simplified and modified the API. Most of the time during the port was spent integrating the existing HaRe architecture to work with GHC's token stream and AST. In particular, this required writing the instances of `Typeable` and `Term` by hand, which is now done automatically by GHC. Integrating GHC with Strafunski was unfortunately a very time consuming task and —due to the unforeseen technical challenges— it was estimated that half the renaming refactoring had been ported to the GHC API, and no major technical challenges were expected in the remainder of the port. If time is available in the future, the HaRe team wishes to continue the work in porting HaRe over to GHC.

2.5 HaRe: The Haskell Refactorer

This section describes some of the most important parts of the Haskell Refactorer, HaRe that are used in this thesis. Like most program transformation tools, HaRe transforms an AST representation of the program, but HaRe also uses a combination of the AST and the token stream to preserve the layout of the refactored program as much as possible. HaRe also includes an API that allows refactorings to be designed on top of the existing infrastructure libraries.

2.5.1 Gathering Information

A Haskell refactoring tool needs the following information to be able to carry out a refactoring:

- The AST representation of programs.
- Scope information and binding information.
- Type information.
- Module information.
- Layout and comment information.

HaRe derives this information from the Programatica front end. The scope and type information is extracted from the Programatica AST. The comments and layout information are taken from the Programatica token stream; in particular, the comments are taken from the token stream generated by the lexer, and the layout information is extracted from the token stream generated using the location information in the AST.

2.5.2 Transforming and Analysing

Most refactorings involve names and their manipulation, and it is crucial that these manipulations do not disrupt the binding structure of the program. Scope information gives a view of the structure of the program; it also makes program analysis and transformation possible.

2.5.3 Preserving Appearance

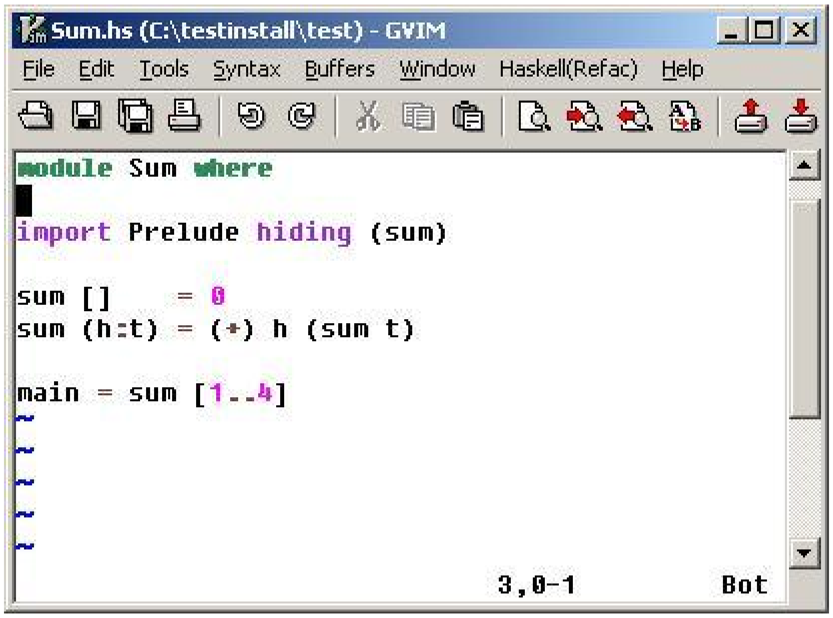
The Kent group made an important decision to preserve the appearance of programs wherever possible. For example, if someone refactors their Haskell code using HaRe, they would expect the refactored program to look as much like the original as possible. The Kent group's approach was to preserve program appearance by making use of both the AST and the token stream. The program analysis is carried out using only the AST, but it is the AST and the token stream that are used to carry out the program transformation. After a refactoring, instead of pretty printing the transformed AST, HaRe extracts the program source from the token stream. However in some cases new code is introduced into the refactored program. In these cases the new code must be pretty printed as there is no previous record of it in the Token Stream. HaRe's solution carries out program analysis with the AST, but it performs program transformation with both the AST and the token stream; whenever the AST is modified, the token stream will be modified to reflect the same change in the program source.

2.5.4 Interfacing HaRe to Text Editors

HaRe can be invoked from either of two program editors: VIm and (X)Emacs, or from the command line. The integration was mainly developed by Reinke. HaRe can also be executed from the command line by running the `pfe` command. It supports a host of commands ranging from basic project management to type checking, simple program transformations, etc. The Kent group extended the Programatica command set with their own refactoring commands. These refactoring commands can be run from the command line just like those from Programatica itself.

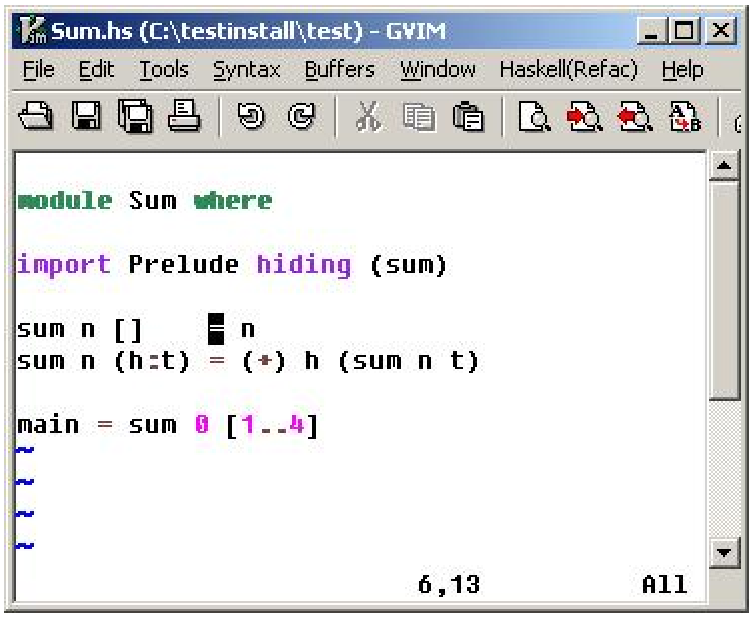
2.5.5 An Example of HaRe

This section shows a brief insight of the HaRe tool in practice. In what follows, we present a sequence of screen shots showing some of the rudimentary refactorings already defined in HaRe. The screen shots are taken from the VIm implementation of the HaRe editor interface, however the same process also applies directly to the (X)Emacs implementation in the same way. The following shows an example of a Haskell program embedded in VIm:


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

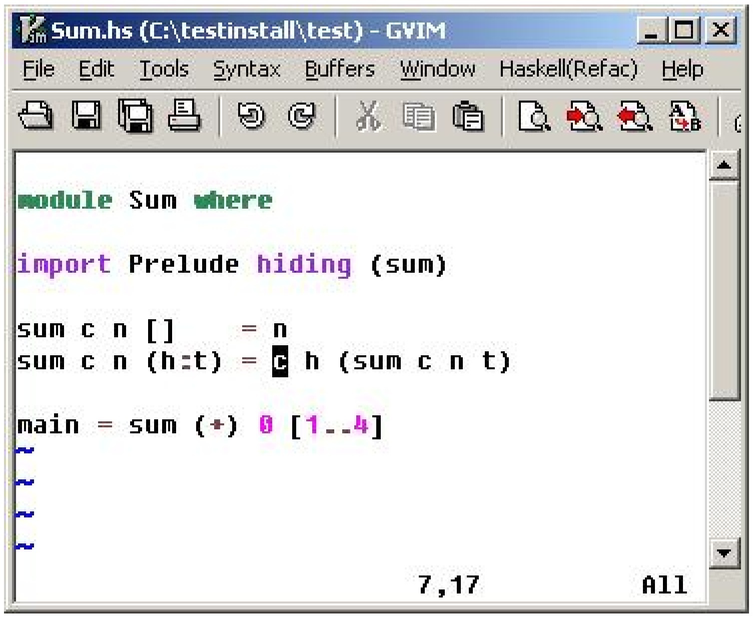
```
module Sum where

import Prelude hiding (sum)

sum []    = 0
sum (h:t) = (+) h (sum t)

main = sum [1..4]
~
~
~
~
~
                             3,0-1          Bot
```

The first step is to generalise over the `0` step of the `[]` case of `sum`. `0` is highlighted, and *generaliseDef* is chosen from the refactorer drop down menu. A prompt is presented asking the user for a new name, and the parameter `n` is entered.


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

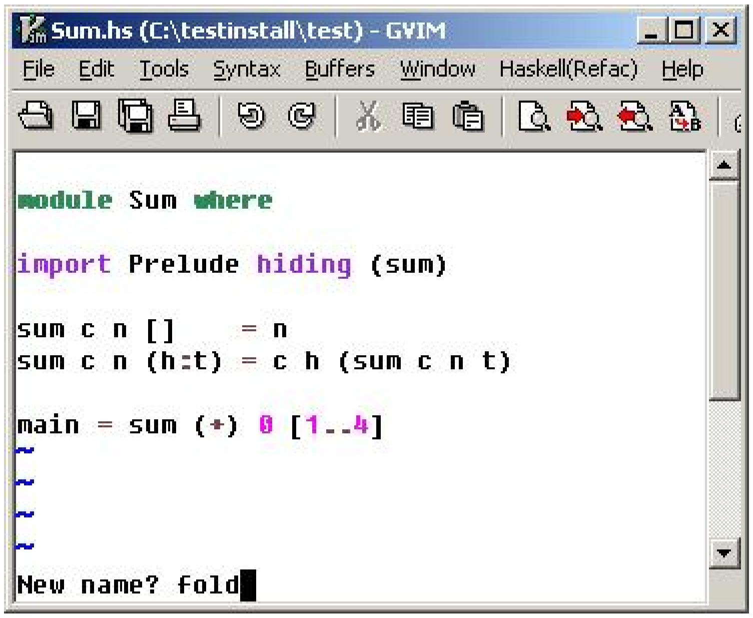
```

module Sum where

import Prelude hiding (sum)

sum n []    = n
sum n (h:t) = (+) h (sum n t)

main = sum 0 [1..4]
~
~
~
~
                             6,13          All
```

The same process is now applied to `(+)` in the second equation of `sum`. This time `c` is entered as the parameter.


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

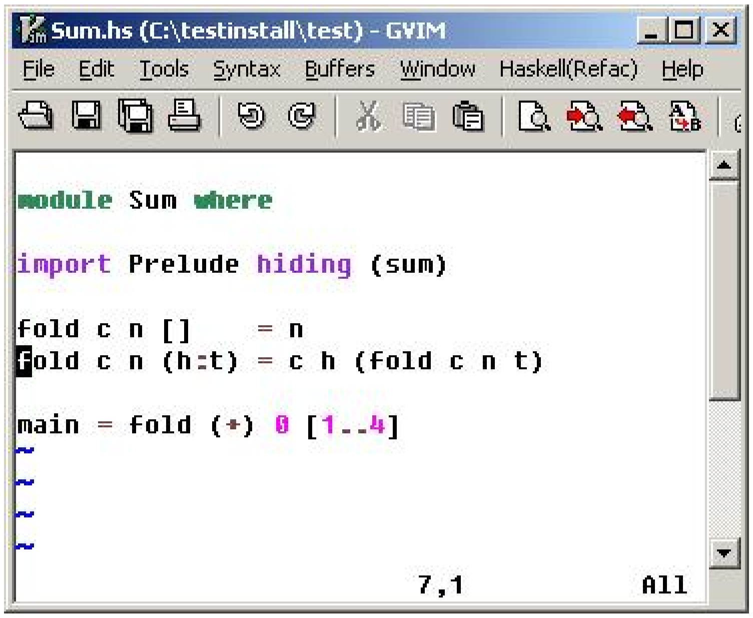
```
module Sum where

import Prelude hiding (sum)

sum c n []     = n
sum c n (h:t)  = c h (sum c n t)

main = sum (+) 0 [1..4]
~
~
~
~
                        7,17          All
```

The function `sum` is now renamed to better reflect its more general definition. The user positions the cursor over the start of `sum` and chooses *rename* from the HaRe drop down menu.


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

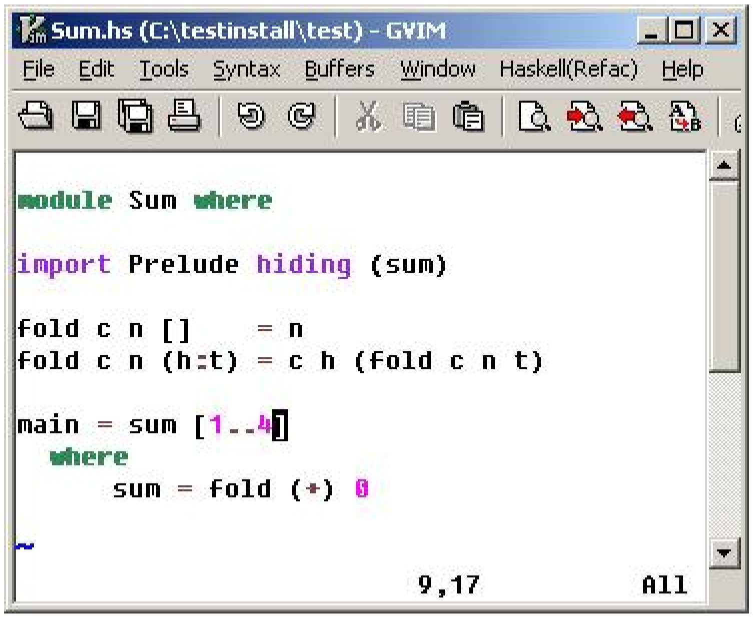
```
module Sum where

import Prelude hiding (sum)

sum c n []     = n
sum c n (h:t) = c h (sum c n t)

main = sum (+) 0 [1..4]
~
~
~
~
New name? fold
```

fold is entered as the new name for the definition:


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

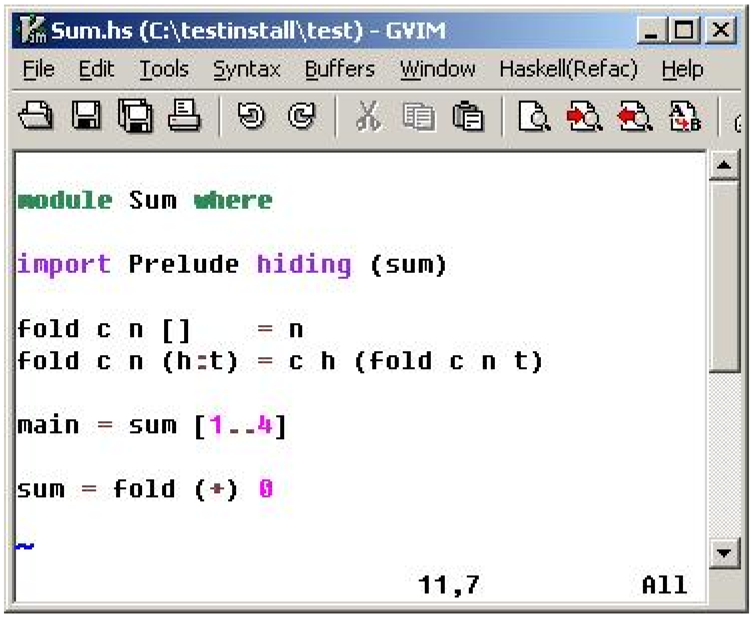
```
module Sum where

import Prelude hiding (sum)

fold c n []     = n
fold c n (h:t) = c h (fold c n t)

main = fold (+) 0 [1..4]
~
~
~
~
                         7,1           All
```

The next step is to introduce a new definition for the expression `fold (+) 0` as it is the exact definition of the original `sum` function. The expression is highlighted, and *introNewDef* is chosen from the HaRe menu with `sum` entered as the new name for the definition:


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help

```
module Sum where

import Prelude hiding (sum)

fold c n []     = n
fold c n (h:t) = c h (fold c n t)

main = sum [1..4]
  where
      sum = fold (+) 0

~
                         9,17           All
```

Since this definition might be reusable, it makes sense to lift it to the top level of the program. The cursor is positioned a the start of `sum` and *liftToTopLevel* is chosen from HaRe:


Sum.hs (C:\testinstall\test) - GVIM
File Edit Tools Syntax Buffers Window Haskell(Refac) Help
11,7
All

```
module Sum where

import Prelude hiding (sum)

fold c n []     = n
fold c n (h:t) = c h (fold c n t)

main = sum [1..4]

sum = fold (+) 0

~
```

2.5.6 An API for Defining Refactorings

An API has been defined to expose HaRe's infrastructure for implementing refactorings. It contains a collection of functions for program analysis and transformation, covering a wide range of syntactic entities of Haskell 98. Moreover, the token stream manipulations, used to ensure that layout and comments are preserved, are hidden in the program transformation functions provided by this API. The refactorings presented in this thesis both extend and make use of the HaRe API.

2.5.7 Refactoring Unit Testing

In order to test the correctness of the refactorings, it was the decision of the Kent group to use the Haskell Unit tester, HUnit [43]; HUnit is a unit testing framework for Haskell, inspired by the JUnit [27] tool for Java. With HUnit, as with JUnit, you can easily create tests, name them, group them into suites, and execute them, with the framework checking the results automatically. Test specification in HUnit is even more concise and flexible than in JUnit, thanks to the nature of the Haskell language. HUnit currently includes only a text-based test controller, but the framework is designed for easy extension.

Tests in HUnit are specified in terms of test cases, which are themselves composed of assertions. In the testing of HaRe, the Kent group have modified the basic HUnit testing platform so that refactorings can be tested by comparing an expected version of the refactored file with the actual refactored file from HaRe. Each refactoring is given a testing directory, with a set of test modules, and a HUnit test case file. The test case file comprises of a list informing HUnit which files to refactor, the refactoring to perform, and any parameters for each file. The list of refactoring tests is made up of positive and negative tests, and require the expected outcome of the refactoring to be stored in a file suffixed with `_TokOut.hs`. Say the file to be tested is `Foo.hs`, the positive tests succeed if the outcome of the refactoring on `Foo.hs` exactly matches the contents of `Foo_TokOut.hs`; similarly, the negative tests fail if the outcome of the refactoring does not match the contents of `Foo_TokOut.hs`. HaRe also performs other kinds of testing, such as testing the identity of the AST representation of the refactored file to the expected AST; this has the advantage of ignoring the layout of Haskell programs.

Testing in this manner has the advantage of being able to keep track of the behaviour of the refactorings during the development process. If a new feature is added, running the test suite on that refactoring will still check to make sure the old behaviour is preserved. Using this testing framework also allows the entire suite of refactorings in HaRe to be tested after a change is made to the API or the Programatica framework.

The drawback, however, is that regression testing does not uncover all possible bugs, and it may be possible that the refactorings miss some specific corner cases. This has been overcome, to a minor extent, by releasing beta versions of HaRe to the public for testing. The Erlang refactorer, Wrangler currently uses Quickcheck to test the correctness of refactorings [47]; it is possible to extend the HaRe testing framework to use QuickCheck [20] instead of HUnit for the testing of refactorings.

2.6 Preserving Correctness

The meaning of preserving correctness for refactorings is as follows: given the same input value(s), the program should produce the same output value(s) before and after a refactoring. It is possible this basic constraint may not be sufficient in some applications, in which case execution time or memory consumption may be added to the criteria.

It is difficult, however, to speculate that a refactoring may make a program more or less efficient as this also depends on the Haskell implementation being used. Haskell is a general purpose programming language, with no constraints on execution time or memory consumption. It is therefore sufficient to use only the basic criterion of correctness when implementing refactorings: given a `main` function in a `Main` module of a program, the function `main` should produce the same output for the same input before and after the refactoring. The semantics of other programs could be changed after a refactoring, as long as these changes do not affect the value of `main`.

A refactoring should have three different aspects: a set of *side conditions* that need to be met for the refactoring to preserve correctness; a set of *transformation rules* that dictate how the refactoring should transform the source code; and a *proof* showing that the transformation preserves the program's correctness by ensuring that the side conditions are met. However, since Haskell has no formally defined set of semantics, writing a proof of correctness is not straightforward; an attempt has been made, however, to prove the correctness of refactorings for a subset of the Lambda Calculus by Li [67]. It could be possible to extend this calculus to apply to the full Haskell 98 standard, and then attempt to write a proof of correctness for each refactoring described in this thesis; this, however, is left as an area of future research. The refactorings presented in this thesis are implemented with Haskell 98 properties that are described in the remainder of this section and the implemented refactorings are designed so that these conditions are always met.

- **Unique binding**. At each use of an identifier within a Haskell program, it must be possible to resolve where in the program the identifier is bound. The binding must take place within the namespace of the program, whether in the body or export list of a module. An *undefined identifier* error will be incurred if no definition is bound to an identifier. An *ambiguous reference* error is given if more than one binding for a top level identifier exists (this can sometimes be resolved with qualified names).

- **No name clashes in the export list**. The unqualified names of the entities exported by the module must all be distinct to avoid name clashes.

- **No unexported entities in import declarations**. The entities explicitly stated in an import declaration that imports a module, say `M`, must also be exported explicitly by `M`.

- **Distinct entity names**. The entity names declared in the same scope and namespace must be distinct from each other, otherwise a *name conflict* error will be incurred. However, the same name can be declared in inner or outer scopes, provided that the entity name does not conflict with other bound names within that scope.

- **Compatible type signature**. After the refactoring, the type signature should be compatible with the type inferred by the compilers for the related entity.

- **No name capture**. Name capture must not happen during the refactoring process. Name capture occurs when an identifier that should be free in a scope becomes bound because of the declaration of the same name across nested scopes. Suppose we select `foo` at a top-level scope, if the sub-expression instance under refactoring also has a local definition called `foo`, the result of the refactoring will result in a call to the wrong `foo` (and will result in unexpected behaviour). Consider, for example, the following:

```
showAll = (concat . format) foo
          where
            foo = map show


foo = (concat . format)
```

Shadowing is a term used to denote when a variable in an inner-scope is no longer referenced due to a variable of the same-name defined in an outer-scope.

Replacing `(concat . format)` in the definition of `showAll` with a call to `foo` will give an incorrect definition of `showAll`, as `foo = (concat . format)` shadows the local definition of `foo` within the `where` clause of `showAll`. The fold is possible, however, if `foo` is defined in a `let` expression within a `where` clause, so long as the scope of this did not include the sub-expression that is selected for folding. For example, it may be possible to define:

```
showAll = (concat . format) myMap
          where
            myMap = let foo = map show in foo


foo = (concat . format)
```

It is then possible to replace `(concat . format)` with a call to the correct `foo`.

2.6.1 The Left and Right Inverse of Refactorings

Some of the refactorings presented in this thesis are implemented as a pair, where both refactorings are described as the inverse of each other. Notably, these refactorings are: add and remove a constructor (described in Section 5.2 on Page 111) and add and remove a field to a data type (described in Section 5.4 on Page 118). In this section we describe what it means for a refactoring to be an inverse of another refactoring, and how the refactorings in this thesis are not true inversions. One way of thinking about this is to disentangle the notions of inverse and left and right inverses.

We say that `f:X -> X` and `g:X -> X` are inverse if `f . g` and `g . f` are both the identity on `X`. In other words `f(g(x))=x` and `g(f(x))=x` for all `x` in `X`.

There is also a weaker notion: we say that `f` is a left inverse of `g` (or equivalently that `g` is a right inverse of `f`) if for all `x`

```
f(g(x))=x
```

Suppose that `X = Integer` and that

```
g(x) = 2*x
f(x) = x div 2
```

Then for all `x`

```
f(g(x)) = (2*x) div 2 = x
```

but it is not true that `f` and `g` are inverse, since

```
g(f(3)) = 2*(3 div 2) = 2*1 = 2
```

If `adding` takes a Haskell 98 program and transforms it into another Haskell 98 program, it is possible to define its type as follows:

```
adding :: Program -> Program
```

Adding a constructor adds an entity (either a constructor or a field of a constructor) to a selected data type and adds default `error` code throughout the program where it is possible that a call to the new entity may occur. We can say that adding is *total*: we can apply it to any program where the new name of the constructor (or field) will not conflict with an identifier in the same scope of the program. The true inverse, therefore, must be a *partial* function: only being applied to the cases where an entity has been added (removing the default behaviour) and in cases where the identified entity is not being used by the program.

In this thesis however, it was decided that the true inverse was not pragmatic to the programmer, and instead, a *destructive* approach was implemented. This destructive approach removes any selected entity (a constructor or data type field), including those entities being used by the program; this is done by introducing default `error` behaviour for identified occurrences of the entity throughout the program. The destructive approach is not a true inverse as the domain it can be applied to is *larger* than the domain for adding. Therefore, we can say that removing is a left inverse, since

```
removing . adding = id
```

but not the reverse.

2.7 Implementing Refactorings

Whilst refactorings differ from each other in their side-conditions and transformation rules, their implementation normally follows a similar pattern. In this section, we describe how the *fold as patterns* refactoring was implemented, in complete detail, using real code from HaRe. *Folding as patterns* is one of the more basic, but useful, refactorings supported by HaRe and described in Section 4.3 on Page 85. To show a simple example of the refactoring in practice, consider the following:

```
g [] = []
```

```
g ((f,r,c,Module):rest) = (f,r,c,Module) : g rest
```

It makes sense to convert `(f,r,c,Module)` on the left-hand-side of `g` into an as-pattern:

```
g [] = []
g (a@(f,r,c,Module):rest) = a : g rest
```

The fundamental requirement for this refactoring is not to violate the binding structure of the program. Basic as it is, the implementation is by no means a trivial task due to the complex binding nature of Haskell programs.

A refactoring normally involves the following steps:

- Transform the program source into an internal representation; this is currently done automatically by Programatica, where it transforms the source code into an AST and a token stream.
- Locate the focus of the refactoring. Usually the user is required to highlight a particular function or expression in the program. For example in folding as patterns, the user is required to highlight a specific pattern match of interest. The pattern match in question is located within the AST using location information supplied by the text editor.
- Validate the side conditions of the refactoring.
- Perform the refactoring transformation. Currently the AST and token stream are modified directly by HaRe. Although refactorings modify the AST directly, the token stream is modified indirectly by the HaRe API.
- Present the refactored program to the user. This requires the token stream to be pretty printed and written to the original file, the editor automatically refreshes the buffer, and the refactored program is presented.

These steps are typically implemented in the top-level function of the refactoring. The top-level function for folding as patterns is shown in Figure 5 on Page 51.

```
refacAsPatterns args
  = do let
           fileName    = args!!0
           name        = args!!1
           beginRow    = read (args!!2)::Int
           beginCol    = read (args!!3)::Int
           endRow      = read (args!!4)::Int
           endCol      = read (args!!5)::Int

       AbstractIO.putStrLn "refacAsPatterns"
       unless (isVarId name)
             $ error "The new name is invalid!"

       -- Parse the input file.
       modInfo@(inscps, exps, mod, tokList) <- parseSourceFile fileName
       let exp = locToExp (beginRow, beginCol) (endRow, endCol) tokList mod

       case exp of
         Exp (HsId (HsVar (PNT (PN (UnQual "unknown") (G (PlainModule "unknown")
         "--" (N Nothing))) Value  (N Nothing)) ))
           -> do

              let (pnt, pat, match)
                  = findPattern tokList (beginRow, beginCol) (endRow, endCol) mod
              ((_,m), (newToks, newMod)) <- applyRefac
                       (changePattern name pat match) (Just (inscps, exps, mod, tokList))
                       fileName
              unless (newMod /= mod)
                                 $ AbstractIO.putStrLn "Pattern does not occur on the rhs!"
              writeRefactoredFiles False [((fileName, m), (newToks, newMod))]
              AbstractIO.putStrLn "Completed."

         _
           -> do
               let pat = getPat exp mod
               ((_,m), (newToks, newMod)) <- applyRefac (changePatternSimple name pat exp)
               (Just (inscps, exps, mod, tokList)) fileName

               writeRefactoredFiles False [((fileName, m), (newToks, newMod))]
               AbstractIO.putStrLn "Completed."
```

Figure 5: The top-level function of folding as patterns

The formal argument to `refacAsPatterns`, namely `args`, is a parameter passed into the refactoring by the editor, and it is a `String` containing the filename, the name for the as pattern to be introduced and the source position information. The first step of the refactoring, therefore, is to retrieve this information from `args`. After this, the new name is checked to make sure that it is a valid name for a Haskell identifier. Specifically, a call to the HaRe API function `isVarId` is used; a call to `error` is performed if this side condition fails. The next step is to parse the source file; in order to do this, a call to Programatica's parser is performed and returned is a list of in-scope names (`inscps`), a list of export names (`exps`), the AST representation of the program (`mod`) and the token stream (`tokList`).

In this refactoring, a user can select either a pattern match occurring on the left hand side of an equation, or an expression occurring on the right hand side of an equation. If the user selects the pattern match, all instances of it on the right hand are transformed into an as pattern call; if an expression is instead selected, only that instance is converted into as pattern. Therefore, this refactoring needs to be able to distinguish between the *for one* mode and the *for all* mode. To do this, the refactoring first checks to see if an expression is selected; this is done with a call to the API function `locToExp`.

A `case` analysis is then performed over the expression returned by `locToExp`. If the program source at the position passed into `locToExp` is not an expression, then a *default* expression is returned. The `case` expression checks to see whether or not this default expression was indeed returned by `locToExp`, and if it was, the refactoring instead attempts to retrieve the selected pattern match, with a call to `findPattern` within the first branch of the case statement.

The function `findPattern` as shown in Figure 6 on Page 54 traverses the AST for the instance of the pattern match selected by the user. The function uses a type-unifying strategy to retrieve the pattern match instance. If the instance cannot be found, the function performs a call to `error` terminating the refactoring. Upon success, the construct that the pattern match is bound to is returned wrapped in a `Patt` type (shown in Figure 9). The purpose of `Patt` is to distinguish between different Haskell constructs that a pattern match can occur in. For example, in Haskell, pattern matching can occur on the left hand side of an equation (specified by `Match`); within a `case` expression (specified by `MyAlt`); within a pattern matching occurring within a `do` expression (specified by `MyDo`); and within a pattern matching a list comprehension (specified by `MyListComp`). The actual refactoring is performed within a `applyRefac` context; due to the modification of the AST and token stream, HaRe is currently configured to refactor within a state monad, and this makes `IO` particularly difficult in some refactorings. If the program entity is a pattern match, `changePattern` is the refactoring that is called, otherwise `changePatternSimple` is called instead. There is an additional side condition if a pattern match is selected: the pattern match must occur on the right hand side of the equation, if it does not, then an error is returned and the refactoring is terminated. This check is performed by comparing the modified AST against the original: if the pattern match does not occur on the right hand of the equation, then the AST will be returned unmodified. Finally, the refactored AST and token stream are pretty printed to a file, via `writeRefactoredFiles`.

Next, we describe the functions `changePatternSimple` and `changePattern`. `changePatternSimple` is called when the user selects a valid expression and has the intention that only that expression must be converted into a call to an as pattern. Figure 7 on Page 55 shows the implementation for `changePatternSimple`. The implementation of the function is typical of the implementation of a refactoring in HaRe; in particular, `changePatternSimple` takes a number of arguments: the `name` of the as pattern to be introduced, the identified pattern match (`pat`) and the highlighted expression (`exp`). The last parameter is a tuple containing the in-scopes, the export relations of the module and, most importantly, the AST itself (`t`). The token stream is passed into the function implicitly and is updated by the monadic properties of the HaRe state monad. One of the side conditions of the refactoring, is that the introduced as pattern name cannot already exist in the scope of the selected equation, if it does, a call to error is performed and

```
findPattern toks beginPos endPos t
  = fromMaybe (defaultPNT, defaultPat, error "Invalid pattern selected!")
              (applyTU (once_tdTU (failTU `adhocTU` inMatch
                                          `adhocTU` inCase
                                          `adhocTU` inDo
                                          `adhocTU` inList )) t)

    where
       --The selected sub-expression is in the rhs of a match
      inMatch (match@(HsMatch loc1  pnt pats rhs ds)::HsMatchP)
        -- is the highlighted region selecting a pattern?
        | inPat pats == Nothing = Nothing
        | otherwise = do
                         let pat = fromJust (inPat pats)
                         Just (pnt, pat, Match match)

      inCase (alt@(HsAlt s p rhs ds)::HsAltP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                          let pat = fromJust (inPat [p])
                          Just (defaultPNT, pat, MyAlt alt)

      inDo (inDo@(HsGenerator s p e rest)::HsStmtP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                          let pat = fromJust (inPat [p])
                          Just (defaultPNT, pat, MyDo inDo)
      inDo x = Nothing

      inList (inlist@(Exp (HsListComp (HsGenerator s p e rest)))::HsExpP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                          let pat = fromJust (inPat [p])
                          Just (defaultPNT, pat, MyListComp inlist)
      inList x = Nothing

      inPat :: [HsPatP] -> Maybe HsPatP
      inPat [] = Nothing
      inPat (p:ps)
       = if p1 /= defaultPat
           then Just p1
           else inPat ps
              where
                p1 = locToLocalPat beginPos endPos toks p
```

Figure 6: The function that finds a pattern match using Strafunski

```
changePatternSimple name pat exp (_, _, t)
 = do

       inscopeNames <- hsVisibleNames exp t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat exp

       newT <- update exp newExp t
       newT2 <- update pat convertedPat newT
       return newT2
```

Figure 7: The function to change an expression instance to an as pattern

the refactoring is terminated. `hsVisibleNames` —a HaRe API function— is used to return a list of all names in the scope of the highlighted expression; it is important to note that location information is removed from `inScopeNames` so that name capture can be calculated up to alpha equivalence. The implementation of `hsVisibleNames` is given in Appendix H.

`pat` is converted to an expression, and if the highlighted expression matches the pattern, the highlighted expression is transformed into a name. This is done with a call to `update` (a HaRe API call, defined in Appendix H) ; an additional `update` is then called to convert the pattern match into a pattern match with an as-pattern.

The function `changePattern` is used if the user performs the refactoring by selecting a pattern match rather than an expression. This places the refactoring in a *for all* mode: every instance of the pattern match on the right hand side of the equation is replaced with a call to an as pattern. Figure 8 on Page 56 shows the implementation for `changePattern`. Following the template of `changePatternSimple`, `changePattern` also takes the as pattern `name` and AST (`t`) as a parameter. In addition to these, `changePattern` also takes one of the following constructs as defined by `Patt` in Figure 9 on Page 56.

```
changePattern name pat (Match match) (_, _,t)
  = do
       newDecl <- lookInMatches t name pat [match]
       newT <- update match (newDecl !! 0) t
       return newT

changePattern name pat (MyAlt alt) (_, _, t)
  = do
       newAlt <- lookInAlt t name pat alt
       newT <- update alt newAlt t
       return newT

changePattern name pat (MyDo inDo) (_,_, t)
  = do
       newDo <- lookInDo t name pat inDo
       newT <- update inDo newDo t
       return newT

changePattern name pat (MyListComp inList) (_,_,t)
 = do
       newList <- lookInList t name pat inList
       newT <- update inList newList t
       return newT
```

Figure 8: The function to change all expression instances to an as pattern

```
data Patt = Match HsMatchP | MyAlt HsAltP | MyDo HsStmtP | MyListComp HsExpP
             deriving (Show)
```

Figure 9: The data type to distinguish different places a pattern match could occur

The principle for `changePattern` is the same for transforming all constructs in Haskell where a pattern match may occur, the only difference is how the AST is transformed. A description of how `changePattern` works for pattern matching occurring on the left hand side of an equation (specified by `Match` in the figure) is given here for reasons of brevity. A complete listing of the implementation of folding as patterns is given in Appendix G. In Figure 8 on Page 56 the implementation for `changePattern` working over `Match` constructs is shown. The implementation is simple: a new equation is generated with a call to `lookInMatches` and the original equation (`match`) is transformed into the new equation by a call to `update`.

The function `lookInMatches` (defined in Figure 10, on Page 58) has to take into account equations with guards and without guards. If there are guards, `lookInMatches` must also traverse into the guards to check for instances of the pattern match and convert those instances into the as pattern name. Much of the implementation of `lookInMatches` has similar functionality to `changePatternSimple` as described above. The new name for the as pattern is checked to see whether or not it occurs in the scope of the equation. If it does not then the right hand side of the equation is traversed for instances of the pattern match (by a call to `checkExpr`). The equation is transformed into a new equation with the transformed expression (and guards, if applicable) and returned by the function; this return value is then updated into the AST and token stream.

2.8 Summary

This chapter has given an overview of the framework on which HaRe is built. The technology used, namely Programatica, Strafunski and GHC, form the core framework for implementing the refactorings in this thesis. An overview of Programatica was given in Section 2.2; the Strafunski generic tree traversal library was discussed in Section 2.3; GHC was discussed in Section 2.4. An overview of HaRe was given in Section 2.5; finally, the chapter concludes with giving a worked

```
lookInMatches _ _ _ []  = return []
lookInMatches t name pat (match@(HsMatch s pnt (p:ps) (HsGuard g@((_,_,e):gs)) ds):ms)
    = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat (p:ps)
       let convertedPat = convertPat name pat
       newGuard <- checkGuards convertedPat g
       newDS <- mapM (convertPatterns t name pat) ds
       rest <- lookInMatches t name pat ms
       if newGuard == g && newDS == ds
         then do
           return (HsMatch s pnt (p:ps) (HsGuard g) ds : rest)
         else do
           return (HsMatch s pnt newPats (HsGuard newGuard) newDS : rest)


lookInMatches t name pat (match@(HsMatch s pnt (p:ps) (HsBody e) ds):ms)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat (p:ps)
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- mapM (convertPatterns t name pat) ds
       rest <- lookInMatches t name pat ms
       if newExp == e && newDS == ds
         then do
           return (HsMatch s pnt (p:ps) (HsBody e) ds : rest)
         else do
           return (HsMatch s pnt newPats (HsBody newExp) newDS : rest)
```

Figure 10: The function to transform equations for as pattern folding

example of an actual refactoring that was implement in HaRe in Section 2.7.

Chapter 3

Program Slicing

This chapter presents a series of refactoring related to program slicing, and in particular a new definition of program slicing is given for Haskell programs. Section 3.2 introduces dead code elimination; Section 3.3 defines what program slicing means for Haskell programs in the form of Splitting (defined in Section 3.3.1) and Merging (defined in Section 3.3.2).

3.1 Introduction

Weiser, in [120], introduces a program slice S as a *reduced executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P*. This process is driven from a *slicing criterion*, usually (a variable representing) the line number and expression of interest, which is used to represent the point in the code whose impact is to be observed with respect to the entire program. Weiser introduced the concept that is now known as *backwards, static* slicing. A backwards slice consists of all parts of a program that have an effect on the criterion in question. Another form of program slicing is a forwards slice [109]. Starting with the slicing criterion, or the program point of interest, a forwards slice is made up of all parts of the program that the criterion will potentially affect.

Orthogonal to the choice between forwards and backwards slicing is the distinction between *static* and *dynamic* slicing:

- Static slicing creates a program slice *given no more information* than which variables we are interested in.
- Dynamic slicing relies on additional available information in order to give a much smaller slice than static slicing. Specifically, dynamic slicing typically requires knowledge of one particular program execution. One of the program's inputs must be given in order for the dynamic slice to continue.

Static, backwards program slicing in an imperative language, is concerned with extracting the computation required to compute a particular variable or program statement. A natural analogue to this within the functional paradigm is a subset of the components of a structured result, for example, the fields of a tuple in Haskell. Selecting the components of a result and looking at the computation corresponds to backwards slicing, whereas selecting the subset of the arguments of a function, and seeing what can be computed on the basis of them alone, corresponds to a forwards slice. This chapter presents a number of this style of slicing-based refactorings for Haskell.

The chapter starts with dead code elimination and based upon that the process is extended to introduce a notion of function splitting and merging. As an example of merging and splitting, consider the following Haskell library functions, `take` and `drop`:

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ []= []
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"
```

```
drop :: Int -> [a] -> [a]
drop 0 xs            = xs
drop _ []            = []
drop n (x:xs)
  | n>0  = drop (n-1) xs
drop _ _ = error "PreludeList.drop: negative argument"
```

As a concrete example of the usage of `take` and `drop`, consider:

```
> (take 10 "hello world", drop 10 "hello world")
> ("hello worl","d")
```

A merge refactoring allows the creation of a function which provides both results using only one list traversal rather than one traversal for each of `take` and `drop`. In the following example, `splitAt` is the result of merging `take` and `drop`:

```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:ys,zs)
    where
    (ys,zs) = splitAt (n-1) xs
splitAt _ _ = (error "PreludeList.take: negative argument",
               error "PreludeList.drop: negative argument")
```

The following is an example of the usage of `splitAt`:

```
> splitAt 10 "hello world"
> ("hello worl","d")
```

Merging is actually known as *tupling* in the field of program transformation, and was originally proposed by Pettorossi [90], as a strategy for composing efficient computations by avoiding repeated evaluations of recursive functions. It is defined and implemented for Haskell here for completeness and for historical purposes.

Splitting is the converse of merging, for example, the extraction of the definitions of `take` and `drop` from `splitAt`. Although `splitAt` is a predefined Haskell function, it serves as a useful example to illustrate splitting and merging. The process of these refactorings is described in section 3.3.

The following sections first introduce the idea of eliminating code within a function that is not needed; this has two flavours: unused code elimination and irrelevant code elimination. A notion of program slicing in Haskell is then outlined. A backwards, static program slicing technique that takes apart tuple-returning functions is defined. Following this the converse of the splitting operation, namely merging is also defined. The chapter concludes by looking at possible future developments in program slicing for Haskell.

3.2 Code Elimination

Dead code elimination [41] is a compiler optimization used to reduce a program's size by removing the parts of the program which are not needed. Dead code is code that is unreachable by running the program.

This section introduces two flavours of code elimination: dead code elimination and irrelevant code elimination. Dead code elimination is concerned with taking a particular top-level function of interest, and removing any nested declarations within that function that are not needed. Irrelevant code elimination is a generalization of the former and focuses upon removing nested declarations that are not needed to compute a particular sub-expression of interest within the top-level declaration. Irrelevant code elimination can be used to aid the programmer in debugging code.

3.2.1 Dead Code Elimination

Programming "rapidly and badly" and subsequent modifications can cause lots of unnecessary declarations in the program; declarations that are never called and are hence "dead". For example, the code below takes a list of type variables and produces a data structure containing the list of type variables:

```
createApp [var]
 = (Typ (HsTyVar (nameToTypePNT var)))
createApp (v:vars)
 = Typ (HsTyVar (nameToTypePNT (v ++
                                   (concatMap (" -> "++) vars))))
    where
     myConcat :: [ String ] -> String
     myConcat [] = []
     myConcat (x:xs) = (x ++ " -> ") ++ (myConcat xs)
```

The function `createApp` contains one declaration that is not needed: `myConcat` can be removed, as it is not needed to evaluate the result. Dead code can make a function look particularly messy and depending on the particular compiler used may actually consume memory and slow down execution. Dead Code Elimination is a refactoring implemented in HaRe that searches the AST for the definition of a particular function, removing any declarations contained within that particular function that are not needed. The examples provided in this section present only a `where` clause; a similar approach is used to deal with lambda definitions and `let` clauses in the implemented refactoring.

There are two stages to the refactoring, an analysis stage that collects all the information required to do the modification, and a modification stage that actually performs the modification on the AST with the information provided by the analysis stage. Currently the refactoring only removes dead code from within one function. However, it could easily be expanded to take a whole module of functions (or, indeed, a set of modules) into consideration. A refactoring that checks for dead code in all definitions of a Haskell project could also be easily implemented. This would be a composite refactoring built on top of the atomic refactoring for one definition.

In order to use this tool, firstly the user selects a function from the editor window. The user then selects *dead code elimination* from the HaRe drop-down menu. To capture the entire namespace, the whole program is parsed into an AST and token stream. The AST is then traversed using Strafunski to find the particular function in question. This traversal is performed using the location information in the AST; it is possible to traverse into any function corresponding to the selected region in the editor. Once the function is found the function's `where` clause and right-hand-side are retrieved. The function's right-hand-side is then traversed until the result expression is reached. For example the result of `createApp` is:

```
Typ (HsTyVar (nameToTypePNT (v ++ (concatMap (" -> "++) vars))))
```

Declarations declared in the scope of `createApp` must be analysed to check whether they are dead; these declarations can appear within `let` clauses and lamdba expressions within the right-hand-side of the function. Within a lambda expression, patterns that are declared but are not used in the expression are removed. The refactoring takes into consideration nested declarations; for example: `where/let` clauses. Once the result is reached, the free and bound names within the subexpression are calculated. The list of free names is then used to remove those declarations residing on the right-hand-side that do not appear within the list of free names. For example:

```
f x =  z + res              f x =  z + res
       where                       where
       res = f (x-1)               res = f (x-1)
       res2 = f (x+1)              z = 46
       y = x + 1
       z = 46
```

The result expression is `z + res` and the only free variables are `z` and `res`; therefore, `y` can be removed from the right-hand-side of `f` as it is not used within

```
f x = z + res            f x = z + res
      where                    where
      res = f (x-1)            res = f (x-1)
      res2 = f (x+1)           y = x + 1
      y = x + 1                z = 46 + y
      z = 46 + y
```

Figure 11: Removing dead code from a function

`z + res` or in the definitions of `z` and `res` (or their dependents).

Any mutually recursive declarations must be taken care of by ensuring that all free variables in those declarations are retained. For example in the code in Figure 11 it can clearly be observed that the declaration `z` depends on the declaration `y`.

Once the right-hand-side of `f` within Figure 11 has been modified to remove the declarations that are not used, the `where` clause of the function in question is then analysed. This time the free variables are calculated for each sub-expression within the modified right-hand-side; each member of the `where` clause that appears in the list of free variables is then analysed for its free variables. All the declarations in the `where` clause that are not needed by the right-hand-side of the function in question, and do not appear in the dependancy graph of any needed declaration, are removed. After the AST has been modified, the source code is also modified to mirror the changes of the refactoring.

A popular technique in abstract interpretation [22] is strictness analysis [84]. Dead code elimination is related to strictness analysis in that strictness analysis searches for parts of a program that will always be used. Dead code elimination searches for parts of the program that will never be used. Strictness analysis works by abstracting away from the program so that some dynamic information can be inferred in a static way: inferring that the boolean conditional in an `if` expression is `False`, say, and therefore calculating that the consequence of the `if` is never evaluated. This chapter takes a purely static approach to determining dead code.

3.2.2 Irrelevant Code Elimination

Irrelevant code elimination is useful for debugging purposes and to some extent is used in algorithmic debugging [104]. In algorithmic debugging the debugging tool asks the user a series of questions about whether a particular sub-expression in the code is generating the correct result or not. As the questions proceed, the particular parts of the program that the debugging tool is asking questions about becomes more clearly focused. It is often the case, however, that the programmer will have some intuition where the bug will lie within the code. Extracting a particular sub-expression that is suspected to cause a bug increases the speed of fixing the error. Parts of a function that are known (or at least assumed) to be correct are temporarily removed so the programmer can concentrate effort on fixing the incorrect sub-expression.

As described above, the Dead Code Elimination technique may be generalized to facilitate debugging. It is possible to select a particular sub-expression of interest and to have the function pruned of declarations that are not needed by that particular sub-expression. The expression on the right-hand-side is replaced with the selected sub-expression. This particular generalization of dead code elimination is not a refactoring, it is in fact a transformation since it changes the semantics. For example, consider:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )


pad :: [ [a] ] -> [ [a] ]
pad lists = map (pad’ (count lists)) lists
     where
      pad’ count entry = entry ++ (replicate count (head entry))
```

Suppose the programmer suspects there is a bug in `pad`, specifically, the programmer believes that the bug is in the call `count lists`. Isolating out only the call to `count lists` into a new function would allow the programmer to test that call explicitly, eliminating the parts of `pad` that the programmer believes to be correct:

```
count :: [ [a] ] -> Int
count (l:list) = maximum ( map length list )


pad2 :: [ [a] ] -> Int
pad2 lists = count lists
```

The programmer can then place a call to `pad2` in the code, test the program, discover that the formal parameter to `count` is in fact incorrect and undo the previous transformation and correct the error:

```
count :: [ [a] ] -> Int
count list = maximum ( map length list )
```

The definition of `pad` remains unchanged throughout, and the introduction of `pad` can be undone (using remove a definition) once debugging is completed.

3.2.3 Summary

This section has described two flavours of code elimination for Haskell. The first: *dead code elimination* was concerned with looking at the entire result of a function and removing the parts of the function that were not needed to compute the result. The second: *irrelevant code elimination* was concerned with allowing the user to highlight a particular sub-expression in the result and then remove the parts of the function that are not needed to compute the sub-expression. Dead code elimination and irrelevant code elimination, in particular, form the basis of a backwards static program slicing tool.

The next section expands on the work presented here to introduce the notion of *splitting* and *merging*.

3.3 Slicing Based Refactorings

Hitherto, there has been little work on program slicing for functional languages. Ochoa et al.[85] introduced a dynamic slicing technique for a lazy logic language. Rodrigues and Barbosa in their paper [99] describe a program slicing approach for Haskell; the approach described is, unfortunately, just a pruning of a function call graph, and unlike the approach described in this section is not a true program slice for Haskell programs. The Haskell debugger, Hat [17], also includes a form of program slicer, but it is also just pruning a call graph. At this time there is no standalone program slicing tool available for Haskell and therefore a backwards static slicer for Haskell was defined. This section introduces the notion of program splitting and merging. A number of issues with performing splitting and merging are also discussed. This section defines what program slicing means for a functional language. In this section program slicing is introduced and is then implemented as a *splitting* refactoring in section 3.3.1. The converse of this is then described in section 3.3.2.

3.3.1 Splitting

A function may return a structured value, for example, a tuple. The particular examples presented in this section use only pairs, however the technique can easily work over tuples of any order.

Splitting works by selecting an element of the tuple and then working out everything needed to calculate that element. The calculated dependencies are then simply extracted and isolated from the rest of the function.

Splitting is mostly used for debugging purposes. However splitting may also be used to extract functionality from the function so that it can be extended or re-used. The user passes, as a parameter to the splitter, the number of elements of the result of the selected function that are to be extracted.

Consider the function `parseMessage` below. `parseMessage` takes a `String` of messages each separated by the `&` character. `parseMessage` removes the initial message and returns the next message as the first element of the result and the remainder of the message as the second element:

```
type MessageList = String
type Message = String
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
        where
          ys = dropWhile (/= '&') xs
```

As an example of the usage of `parseMessage` consider:

```
> parseMessage "goodbye&hello&world"
> ("hello","&world")
```

The splitter works through each function clause in turn, extracting the elements of the function clauses' result into separate definitions. It is worth noting that the results of these refactorings are themselves ready for a further refactoring, namely replacing `ys` with its definition (removing the `where` clause altogether). It is often the case that further refactorings such as this are performed. The first pattern clause of `parseMessage` is essentially trivial. Therefore the value `[]` is extracted for both elements of the result and two new functions are then created:

```
parseMessage1 :: MessageList -> Message
parseMessage1 [] = []

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
```

It is noted that the two introduced functions, `parseMessage1` and `parseMessage2` can both undergo a renaming refactoring to make their names more meaningful to the programmer. The splitter appends an index to the end of the names of the new functions, if the new names conflict with any other identifier in scope then the splitter chooses a new distinct name. This is simply done by incrementing the index until the name no longer conflicts with another identifier in scope.

The next function clause's result is then analysed. Irrelevant code elimination is then performed for each element in the resulting tuple and the result of the code elimination is placed into new function clauses for `parseMessage1` and `parseMessage2`. `ys` is required by both elements of the result of `parseMessage` so it is retained; the new function clauses are then added to the definitions of `parseMessage1` and `parseMessage2` within the AST.

```
parseMessage1 xs = takeWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs


parseMessage2 xs = dropWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs
```

This gives the new definitions of `parseMessage1` and `parseMessage2`. The source code is then modified to reflect the changes within the AST. The process described here behaves in the same way for any subset of a tuple.

If the function is recursive, splitting acts *generatively*, creating new, recursive, definitions of the split functions. This is analogous to the generative folding work describe in Chapter 4 on Page 82. As an example, consider, again, the definition of `splitAt`:

```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:ys,zs)
    where
```

```
      (ys,zs) = splitAt (n-1) xs
splitAt _ _ = (error "PreludeList.take: negative argument",
               error "PreludeList.drop: negative argument")
```

Extracting just the first element of its result, gives the following definition:

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ []= []
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"
```

`take` is now a new, recursive, definition separate to that of `splitAt`.

3.3.2 Merging

Merging is the process of taking a number of functions and unifying them together into one tuple-returning function. The process described here only merges two functions; obviously the functionality can be easily extended however, to merge together any number of functions.

Merging works by bringing together the code from the selected functions into a new tuple-returning function. Duplicate parts of the function are also removed. Unlike when doing splitting, where names are generated automatically, the user must specify a name for the merged function.

Merging is mostly used to reuse code and improve code efficiency. For example, merging `take` and `drop` into `splitAt` results in only one recursive call instead of two. Merging functions together has the possibility to introduce further code sharing.

The remainder of this section will focus on merging `parseMessage1` and `parseMessage2`.

```
parseMessage1 :: MessageList -> Message
```

```
parseMessage1 [] = []
parseMessage1 xs = takeWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs

parseMessage2 :: MessageList -> MessageList
parseMessage2 [] = []
parseMessage2 xs = dropWhile (/= '&') (tail ys)
        where
            ys = dropWhile (/= '&') xs
```

Firstly, the merge refactoring also checks to see whether the types of the arguments to `parseMessage1` and `parseMessage2` are the same. If the pattern sets are not the same then the merge refactoring cannot correctly unify the patterns; if they cannot be unified the merge refactoring terminates with an error message to the user.

Secondly, the merge refactoring checks to see whether the pattern sets of `parseMessage1` and `parseMessage2` are the same. If they are not then the merge refactoring terminates with an error to the user. Before merging, it is necessary that both functions have the same sets of patterns. If the first function has more pattern clauses than the second for example, then the merge refactoring cannot determine what to place on the right-hand-side when patterns from the first function are not matched by patterns in the second. An error message is presented to the user if the pattern sets are not the same. An additional refactoring was introduced to allow the user to build particular pattern clauses by instantiating general patterns with values of the same type. Consider, for example:

```
f1 0 l = take 42 l
f1 n l = take n l

f2 n l = drop n l
```

Both `f1` and `f2` have general cases defined, however, `f1` has an additional base case defined. For reasons of simplicity, the merge refactoring does not attempt to specialise pattern matching between definitions. Consider, for example:

```
merged 0 l = (take 42 l, undefined)
merged n l = (take n l, drop n l)
```

Each function clause in question is merged together. The first clause of `parseMessage1` and `parseMessage2` both have the same result value. Merging those clauses together is trivial:

```
parseMessage :: MessageList -> (Message, MessageList)
parseMessage [] = ([], [])
```

The second clauses of `parseMessage1` and `parseMessage2` are now considered. The results are unified together and placed into a new function clause:

```
parseMessage xs = (takeWhile (/= '&') (tail ys),
                   dropWhile (/= '&') (tail ys) )
          where
            ys = dropWhile (/= '&') xs
```

The duplicate declaration of `ys` within the `where` clause is then removed. Recursion is also dealt with correctly, consider the example on Page 61 where the functions `take` and `drop` are merged. The recursive calls to `take` and `drop` are replaced with a call to `splitAt` (the merged function name).

3.3.3 Design and Implementation Issues

This section discusses the issues that occurred when designing and implementing the splitting and merging refactorings.

- Sometimes when splitting is used patterns that are the result of recursive calls become redundant. All variables within patterns that are not needed in the result are replaced with wildcards so that no duplications are introduced in the namespace.

- One element of a tuple may depend upon another element. For instance, it is possible for one element to come from the first part of a recursive call, but for it to be returned in the second element in the main body. If one element depends on another, both of these elements must be extracted from the function, otherwise an error occurs.
- Merging and splitting monadic functions is a difficult area, especially if the monad in question is a state monad, for example, the `IO` monad. Merging two `IO` monadic values is problematic because the merge refactoring cannot infer the correct order of sequencing. Side effects also affect splitting in a similar way, for example, an element of the tuple return value may depend upon some data written to a file. It is very difficult for the splitter to determine which parts of the monad can have a potential effect as all expressions have the potential to alter the state. Merging and splitting refactorings on monads will be the subject of future work.

3.4 Summary

This chapter presented a number of refactorings for HaRe. Firstly, a technique was defined to eliminate dead code from Haskell functions and then generalized to remove irrelevant code. A backwards, static program slicer for Haskell was then described as splitting tuple-returning functions; its converse, namely merging, was also described.

Chapter 4

Structural Refactorings

This chapter describes the new structural refactorings that have been defined and implemented in HaRe for this thesis. Following on from the refactoring work by Li [67], and using the refactoring catalogue [96] maintained by Thompson as a basis, a new set of refactorings that complement and extend the existing refactorings in HaRe have been provided. For each refactoring, the following is introduced where appropriate:

- A general **description** of the refactoring, giving an overview of how the refactoring would change the structure of a program. The general description is also intended to give some motivation for the refactoring in question.
- An **example** showing the effects of the refactoring and its inverse operation (performing the refactoring is shown from *above* to *below*; its undo is shown in the example from *below* to *above*). The examples presented in this chapter are merely intended to give an example of how the refactoring performs in its basic sense.
- A set of **comments** detailing the transformation properties of the refactoring. These comments could discuss any general issues that arise from implementing this refactoring, or could discuss potential problems when the refactoring is performed in specific situations.

- A set of **conditions** from *above* to *below*. A refactoring is made up of a set of side-conditions and a set of transformation rules. The side-conditions must be met in order for the transformation to take place, otherwise HaRe terminates with an error message.
- Any **Design Issues** regarding the design of the refactoring. One problem with designing refactorings is the question of compensation or failure. In most cases, the refactorings designed in this thesis take on the philosophy of giving the user what they expect. In some cases this means the refactoring has to compensate by performing smaller implicit sub refactorings. Any of these sub refactorings are detailed in this section. In a similar manner, any issues which cause the refactoring to fail in addition to the conditions are also given in this section.

The following refactorings are described in this chapter:

- Folding (Section 4.1);
- Generative folding (i.e. folding in the style of Burstall and Darlington [14]) (Section 4.2);
- Folding as patterns (Section 4.3);
- Unfolding as patterns (Section 4.4);
- Converting from `let` to `where` (Section 4.5);
- Converting from `where` to `let` (Section 4.6 — the distinction between the two refactorings is also given in the section)
- Case analysis simplification (Section 4.7).
- The issues surrounding the binding of variables in Haskell are given in Section 4.8.
- The chapter concludes in Section 4.9 by discussing the implementation of a symbolic evaluation mechanism.

Before:

```
showAll = (concat . format) (map show)
table  = concat . format
```

After:

```
showAll = table . map show
table   = concat . format
```

Figure 12: Before and after simple fold

4.1 Folding

Folding replaces all sub expressions in the program which are substitution instances of the right hand side of an identified equation with a call to that equation, passing in the instantiations as actual parameters. This refactoring is designed to be the complement of *unfolding* which is described in Li's thesis [67] and can be used to eliminate some duplicate expressions within a program. In addition, folding can also be used to create a name for a common abstraction occurring within the program by abstracting away from a common sub-expression; as long as there is a definition to fold against this can also be seen as naming an abstraction for a common sub-expression (which is indeed what the user has to do before the fold is performed).

4.1.1 Example

An example of folding an instance of the right hand side of a definition, `table`, is shown in Figure 12. In the figure, two definitions are given: `showAll` and `table`. The right-hand-side of `table`, as can be seen, also appears as a sub-expression on the right-hand-side of `showAll`. Folding allows the definition `table` to be selected and all occurrences of its right-hand-side (occurrences within different entities in the same scope as `table`) are replaced with a call to `table`. The right column of the example shows that the sub-expression, `(concat . format)` has been replaced with a call to `table`, passing in `(map show)` as a parameter; this therefore eliminates some duplicated code within the program.

4.1.2 Comments

The fold is not performed within the definition of `table` itself, nor in the body of any function called within the definition of `table`. Performing the fold in the definition of `table` will introduce termination problems, as the fold will create the equation `table = table`. This issue is discussed in more detail on Page 85.

4.1.3 Conditions

If we are folding against a definition `foo` then the side-conditions for folding are:

- Name capture must not happen during the refactoring process. See Section 2.6 on Page 46 for more details regarding name capture.

4.1.4 Design Issues

- If the definition of `foo` contains any pattern matching on its right-hand-side in the form of a case expression, lambda abstraction or `do` notation, then the structure of the clause must match the structure of any potential substitution instances within the program. The variables in the pattern are renamed so that the two compared pattern clauses contain the same binding information. This issue is discussed in more detail in Section 4.8 on Page 103.

- If the selected definition contains guards, then they are converted into an `if..then..else` before the analysis is carried out; this allows the guards to be matched against expressions that would be defined with an `if` statement. For example, consider `foo` is defined:

```
foo x
 | x == 1    = e1
 | x == 2    = e2
 | otherwise = e3
```

this is converted into the following:

```
foo x = if x == 1 then e1 else if x == 2 then e2 else e3
```

which can then be compared against sub-expression instances such as:

```
if y == 1 then e1 else if y == 2 then e2 else e3
```

The problem with transforming guards into an `if..then..else` is that they are evaluated in a different way. Guards belong to the equation level and `if..then..else` expressions belong to the expression level. If there is no `otherwise` clause specified the evaluator proceeds to pattern match against the next equation for the definition being evaluated. However, as an `if..then..else` is an expression, the evaluator has to return a result once the expression has started to be evaluated. Therefore, if there is no `otherwise` defined for the guards, then an `else` clause must be added to the `if..then..else` that returns an `error`, specifying that the pattern match failed this is therefore not a refactoring in some cases. To illustrate this, consider how the following is converted to an `if..then..else`.

```
foo x
 | x == 1    = e1
 | x == 2    = e2
```

As there is no `otherwise`, `foo` is converted into the following:

```
foo x = if x == 1 then e1
   else if x == 2 then e2
   else error "Pattern Match Failure"
```

Converting guards to an `if..then..else` prevents certain cases from being compared; this is a limitation with comparing *equations* against *expressions*.

- If `foo` has any parameters then the call must pass in any relevant instantiations as parameters. If `foo` has any parameters that are not used in its right-hand-side, then `undefined` is passed to those parameters. The redundant parameters are not removed as another refactoring designed to remove parameters from functions has already been defined. If `foo` contains any pattern matching in its parameters, a pattern is created in the call to `foo`, passing in the instantiations —or `undefined`, if applicable— as parameters for the pattern match. To illustrate this problem, consider the following:

```
sumSquares x y = (case (x, y) of
                        (m, n) -> m ^ 2)


sq (n,m) = m ^ 2
```

Suppose we wish to select `sq` and fold any sub-expression instance against its definition, this would replace the expression `m ^ 2` with a call to `sq`:

```
sumSquares x y = (case (x, y) of
                        (m, n) -> sq (undefined, m)


sq (n,m) = m ^ 2
```

- If `foo` contains local definitions —defined either in a `where` clause or a `let` expression, or both— the refactoring only compares simple cases of the form:

```
foo = y where y = e1
```

against `...x...  where x = e1`. This also applies to `e1` being defined by a `let` expression. It would be possible to unfold the definitions in the `where` clause of the definitions first, and then perform a fold over those definitions.

4.2 Generative Folding

Generative Folding replaces an instance of the right hand side of a definition by the corresponding left hand side, creating a new recursive definition. When it was first described by Burstall and Darlington [14], *Generative Folding* was pioneering work in the field of functional program transformation. We use "generative" to denote this special kind of folding that contrasts the folding described in Section 4.1.

To give an example of generative folding, consider the following equation:

```
mapLength xs = map length xs
```

Where `map` is defined as follows:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Unfolding the right hand side of `mapLength` produces:

```
mapLength xs = case (length, xs) of
                    (f, []) -> []
                    (f, (y:ys)) -> f y : map f ys
```

As it can be seen, unfolding the call to `map` on the right-hand-side of `mapLength` introduces a `case` expression to overcome the fact that `map` is defined with multiple pattern matching equations. New variables are introduced in the `case` binding corresponding to variables bound in the pattern matching equations of `map`. Using the original definition of `mapLength xs = map length xs` it is possible to select `map f ys` on the right-hand-side of the unfolded definition, and create a new recursive definition of `mapLength`:

```
mapLength xs = case (length, xs) of
                    (f, []) -> []
                    (f, (y:ys)) -> f y : mapLength ys
```

In the example *symbolic evaluation* as described in Section 4.9 is used to determine that '`f`' is identical to '`length`' through the binding structure of the `case` expression.

4.2.1 Example

An example of this refactoring is shown in Figure 13 on Page 86. The transformation in Figure 13 uses the original definition of `fmt`, namely: `fmt = format "\n"` from above to below to create a new recursive definition of `fmt`. The fold takes place in the unfolded version of the definition. The two versions of the definition of `fmt` cannot coexist in the same Haskell module. The original definition is therefore retained in a comment above the unfolded definition of `fmt`.

4.2.2 Comments

The refactoring here is different in nature to other structural refactorings. The transformation needs to retain the original version of the unfolded definition, which is used to create the new recursive definition. The original definition, `mapLength xs = map length xs`, cannot coexist in the same Haskell module as the unfolded definition of `mapLength` and is therefore removed by the unfolding; in order to perform the *generative fold*, we chose to preserve the definition in a comment, exactly above the unfolded definition:

```
{- mapLength xs = map length xs -}


mapLength xs = case (length, xs) of
                      (f, []) -> []
                      (f, (y:ys)) -> f y : map f ys
```

4.2.3 Conditions

The conditions for a generative fold are as follows:

- The highlighted expression must be symbolically identical (see Section 4.9) the right-hand-side of the equation in the comments. Say the expression to be selected is `f g` and the equation to fold the expression against is defined in a comment, namely: `f = double (1+1)`. `double g` is transformed into `f` if and only if `g` is symbolically identical to `(1+1)`.
- If the equation to be folded against contains an expression with a pattern —such as a *lambda* or a *case* expression— then the structure must match the structure of the pattern in the highlighted expression. This issue is discussed in detail in Section 4.8.

4.2.4 Design Issues

- Say the equation to be folded against is `mapLength xs = map length xs` and the highlighted expression is `map length ys`. The refactoring allows one to transform `map length ys` into `mapLength ys` by renaming `xs` to `ys` in the original equation, as `xs` is a bound variable defined as a parameter to `mapLength`.
- The *generative fold* uses the symbolic evaluation mechanism used for *Case Analysis Simplification* (described in Section 4.9). It is possible to symbolically link `f` with `length` allowing the expression `map f ys` to be selected; and using the equation retained in the comments, it is possible to transform `map f ys` into `mapLength ys`:

```
{- mapLength xs = map length xs -}

mapLength xs = case (length, xs) of
                    (f, []) -> []
                    (f, (y:ys)) -> f y : mapLength ys
```

  The result of the refactoring *generates* a new definition `mapLength` which is recursive and not dependent on `map`.

There is an interesting observation to be made where symbolic evaluation may not be needed. When the same variable occurs in every case of one variable in a tuple case, we can remove it to give that variable. Consider the `case` expression above, removing `f` could result in the following:

```
case xs of
  [] -> []
  (y:ys) -> length y : mapLength ys
```

- This transformation, as is well known from Burstall and Darlington's original work, can lead to a change in the termination properties of the generative definition. In the extreme case, consider folding the equation, `fmt = format "\n"` against itself. This produces the tautological equation: `fmt = fmt`, which whilst itself is a true property of `fmt` fails to give the same definition of the `fmt` function; instead it makes it everywhere undefined.

  Although there has been research into formalising termination checking (Tullsen [112] and Abel [2]), this has not been implemented as part of the refactoring described in this thesis and has been left as an area of future work.

4.3 Folding As-patterns

Folding as-patterns replaces particular sub-expressions on the right-hand-side of an identified equation with calls to an as-pattern, provided that the sub-expression refers to a pattern binding that is defined in scope. For example, consider the following definition:

```
g [] = []
g ((f,r,c,Module):rest) = (f,r,c,Module) : g rest
```

it makes sense to convert '`(f,r,c,Module)`' on the left-hand-side of `g` into an as-pattern:

Before:

```
format :: [a] -> [[a]] -> [[a]]
format sep xs
  = if len xs < 2
       then xs
       else (head xs ++ sep) : format sep (tail xs)

{- fmt = format "\n", after unfolding -}

fmt xs
  = if len xs < 2
       then xs
       else (head xs ++ "\n") : format "\n" (tail xs)
```

After:

```
format :: [a] -> [[a]] -> [[a]]
format sep xs
  = if len xs < 2
       then xs
       else (head xs ++ sep) : format sep (tail xs)

{- fmt = format "\n", after unfolding and
   generative fold against the original definition. -}

fmt xs
  = if len xs < 2
       then xs
       else (head xs ++ "\n") : fmt (tail xs)
```

Figure 13: Before and after generative fold

```
g [] = []
g (lp@(f,r,c,Module):rest) = (f,r,c,Module) : g rest
```

and then transform the expression '`(f,r,c,Module)`' on the right-hand-side into a use of that as-pattern:

```
g [] = []
g (lp@(f,r,c,Module):rest) = lp : g rest
```

The defining location of the pattern is updated with an as-pattern. This refactoring aims to improve the program's efficiency so that complex expressions do not have to be reconstructed from their component parts by introducing sharing. Modern Haskell compilers (such as GHC) may do this optimization automatically, but it is certainly a worthwhile refactoring to limit duplicated code and encourage code reuse.

4.3.1 Comments

The folding "as-patterns" refactoring has two modes of operation: a *for all* mode which allows all references to a highlighted pattern binding to be replaced with a call to an as-pattern; and a *for one* mode which allows an identified pattern occurring within an expression to be replaced with a call to an as-pattern as defined by the user. It is interesting to note that this is a general phenomenon in refactorings.

The *for all* mode allows the user to highlight the pattern of interest so that all occurrences of the pattern in scope are replaced with a call to an as-pattern

The *for one* mode converts a particular highlighted pattern on the right-hand-side of a function and replaces it with a call to an as-pattern.

4.3.2 Example

To show the distinction between a *for all* mode and a *for one* mode of as-pattern folding, consider the examples in Figures 14 and 15 (on pages 89–90). In Figure 14 we select an instance of the constructor application `T tval Empty Empty` on the right hand side of `result`. The refactoring matches this constructor application with the pattern binding in the second argument of `insert` and introduces an as-pattern (the name of the pattern is determined by the user at the runtime of the refactoring); the selected instance of `T tval Empty Empty` is then replaced with a call to the as-pattern (in this case, `t`). In Figure 15 we select the pattern binding `T tval Empty Empty` on the left hand side of `insert` and run the refactoring. The result (shown in the right column) shows that all instances of the constructor application in scope on the right hand side of that particular equation for `insert` have been transformed into calls to the as-pattern. The as-pattern, `t` has also been introduced for the pattern binding in `insert`.

4.3.3 Conditions

Selecting a pattern `Pat x` and introducing `foo` as an as-pattern has the following side-conditions:

- The instance of the pattern must have all of its variables bound to the outer part of the pattern, and not at a local declaration. For example, consider:

  ```
  insert val t@(T tval Empty Empty)
    = T val Empty result where result = (T tval Empty Empty)
                                        tval = 42
  ```

  In the above case, it is not possible to transform the pattern instance `(T tval Empty Empty)` with a call to `t` as a `tval` is already defined in the same where clause.

- The name for the new as-pattern, say `foo`, is introduced by the user when he/she runs the refactoring. The new name must not, therefore, conflict with any other name within the scope of the introduced as-pattern. If the name does conflict the refactoring will not be performed and will instead

Before:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (T tval Empty Empty)
    = T val result2 result where result  = (T tval Empty Empty)
                                 result2 = (T tval Empty Empty)
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

After:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T tval Empty Empty))
    = T val result2 result where result  = t
                               result2 = (T tval Empty Empty)
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

Figure 14: Folding an as-pattern: instance of a *for one* operation

Before:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (T tval Empty Empty)
    = T val result2 result where result  = (T tval Empty Empty)
                                 result2 = (T tval Empty Empty)
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

After:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T tval Empty Empty))
    = T val result2 result where result  = t
                               result2 = t
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

Figure 15: Folding an as-pattern: instance of a *for all* operation

terminate with an error. `foo` must not already be defined in the scope of `Pat x` either as an as-pattern or as a bound variable.

- The refactoring will not let the user perform the refactoring twice. Unless, of course, an undo operation is performed between refactorings.

4.3.4 Design Issues

- The binding location of `Pat x` is converted into `foo@(Pat x)` and the selected occurrence (or all occurrences) of the pattern is replaced with a call to `foo`.
- If the user wants to replace precisely two occurrences of an expression with an as-pattern it is not possible. The user can do the first, but the second will be disallowed on the basis that there is already a pattern variable there. This is an example of where a "for all" is not the same as a "for one" performed repeatedly.

4.4 Unfolding As-patterns

Unfolding as-patterns is the inverse of *folding* an as-pattern (described in Section 4.3). Unfolding replaces a particular variable in a sub-expression that refers to an as-pattern variable, with the actual pattern that the variable is bound to. For example, consider that we have the following definition:

```
g [] = []
g (lp@(f,r,c,Module):rest) = lp : g rest
```

Selecting '`lp`' on the right-hand-side of `g` allows us to unfold the as-pattern, to create:

```
g [] = []
g (lp@(f,r,c,Module):rest) = (f,r,c,Module) : g rest
```

This then allows the unfolded pattern binding to be modified (this will potentially change space behaviour if the variable is repeated).

4.4.1 Comments

Unfolding an as-pattern has two modes of operation: a *for all* mode which unfolds all references to a particular as-pattern in scope; and a *for one* operation which only unfolds a particular pattern occurrence of interest.

The *for one* mode converts an occurrence of an as-pattern reference within an expression into the pattern that is defined by the as-pattern. New names are introduced in the pattern if they are defined as wild cards in the defining location. The "for some" operation is therefore defined by performing the "for one" operation repeatedly.

The *for all* mode allows the user to select the as-pattern binding so that all occurrences of the as-pattern in scope are replaced with calls to the pattern binding.

4.4.2 Example

To show the distinction between a *for all* mode and a *for one* mode, consider the examples in Figures 16 and 17. In Figure 16 we select `t` on the right hand side of `result` and choose to unfold its as-pattern reference. The result, shown in the right column of the Figure, shows that `t` has been replaced with the as-pattern that it refers to, namely: `T tval Empty Empty`. This demonstrates the *for one* operation: as only one instance of `t` has been unfolded. In Figure 17 we select `t` where it occurs as an as-pattern binding in the left hand side of `insert`. The right column, again, shows the result of the refactoring; all references to the as-pattern `t` are replaced with the constructor application of `T tval Empty Empty`.

4.4.3 Conditions

The side-conditions of unfolding `foo` are as follows:

Before:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T val Empty Empty))
    = T val result2 result where result   = t
                                 result2 = t
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

After:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T val Empty Empty))
    = T val result2 result
                  where result  = (T val Empty Empty)
                        result2 = t
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

Figure 16: Unfolding an as-pattern: instance of a *for one* operation

Before:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T val Empty Empty))
    = T val result2 result where result   = t
                                 result2 = t
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

After:

```
data BTree a = Empty | T a (BTree a) (BTree a)
               deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert val (t@(T val Empty Empty))
    = T val result2 result where result   = (T val Empty Empty)
                               result2 = (T val Empty Empty)
insert val (T tval left right)
   | val > tval = T tval left (insert val right)
   | otherwise = T tval (insert val left) right
```

Figure 17: Unfolding an as-pattern: instance of a *for all* operation

- `foo` must be defined as an as-pattern in the immediate scope and its defining location must not shadow any other intermediate declarations of the same name.
- If `foo` is bound to a pattern of the form `Pat x` then there must not be another binding of `Pat x` in the scope of `foo`. The refactoring therefore avoids pattern capture.

4.5 Converting from `let` to `where`

Converting a definition from a `let` to a `where` lifts a definition defined at the *expression* level (a definition in a `let` expression) to a definition at the *equation* level (a definition in a `where` clause). The refactoring widens the scope of the definition and therefore allows it to be used by a wider range of entities defined in the scope of the `where` clause. There is great potential for this refactoring to affect the semantic properties of a program, which is discussed in detail in this section.

4.5.1 Example

In Figure 18 (on Page 98) we select the definition `table` defined in a `let` expression on the right-hand-side of `showAll` for extraction into a `where` clause. The result, shown in the second column of the figure, shows that `table` has been lifted to a `where` clause; a new parameter has also been added to `table`: this is used for passing in the definition of `format` since now the definition of `format` is not visible within the `where` clause. The scope of `table` has now been widened: by allowing the definition `foo` to access it, and indeed, other definitions potentially defined in the `where` clause or the body of `showAll` (including any guards defined in this equation for `showAll`).

4.5.2 Conditions

Supposing the name bound by the definition to be converted is `foo`, the conditions for promoting `foo` from a `let` scope to a `where` scope are as follows:

- Promoting the definition must not intervene between any bindings of `foo` and its uses within the `where` clause.
- No binding for `foo` must exist in the binding group within the `where` clause.
- Free variables that are used by the definition but are not in scope in the `where` clause are passed into the lifted definition through parameters. This requires:

  1. The free variables must not be used polymorphically if the variable is defined by a pattern binding. To show an example of this, consider the following:

     ```
     g r p = let compute = len r + len p;
                 len = genericLength
                  in compute
     ```

     Promoting `compute` to a `where` clause causes the function to raise a type error if the types of `r` and `p` are different. For example, suppose `compute` is indeed lifted to a `where` clause:

     ```
     f r p = let len = genericLength
                  in compute len
                   where
                     compute l = l r + l p
     ```

     We now get a type error when we try to evaluate `f` with the following arguments: `[1,2,3,4]` and `"hello"`. When we lift definitions like this to a `where` clause, we really need to introduce an existentially qualified type, which is not a part of the Haskell 98 standard language. To give an example, suppose that:

```
compute :: Integral b => ([a] -> b) -> b
```

Implicitly the type checker infers that `a` and `b` are qualified as existential types:

```
compute:: forall a . forall b . Integral b =>
                ([a] -> b) -> b
```

The problem is because the `forall a . forall b` is on the left most side, we need to introduce rank-2 polymorphism in all cases, so that `a` can be unified with both the types `[Int]` and `[Char]`:

```
compute:: forall b . Integral b =>
            ([forall a . a] -> b) -> b
```

This is a problem in lambda lifting [52] where we introduce functions (or lambdas) to abstract over a polymorphic entity.

2. The definition of `foo` must be a function binding or a simple pattern binding (i.e. a pattern binding of only a single variable) so that parameters can be passed to the definition if necessary.

4.6 Converting from `where` to `let`

Converting from a `where` to a `let` narrows the scope of the definition by converting a definition defined at the *equation* level (a definition defined in a `where` clause) to a definition defined at the *expression* level (a definition defined in a `let` expression).

4.6.1 Example

Figure 18 (on Page 98) shows an example of moving a definition in a *where* to a definition in a *let*. `table` is selected within the where clause of `showAll` for demotion to a let expression. The result (shown in the left column) shows that `table` is added to the let expression of `showAll`, and the parameter, `format` has

Before:

```
showAll :: Show a => [a] -> String
showAll = let table = concat . format
              format []     = []
              format [x]    = [x]
              format (x:xs) = (x ++ "\n")
                                : format xs
                 in table . map show
                   where
                      foo = concat . map
```

After:

```
showAll :: Show a => [a] -> String
showAll = let format []     = []
              format [x]    = [x]
              format (x:xs) = (x ++ "\n")
                                : format xs
        in (table format) . (map show)
       where
         foo = concat . map
         table format = concat . format
```

Figure 18: Conversion of a `let` to a `where` and vice versa

been removed (`format` is now in scope) — detecting this relies on the binding information of variables in the AST.

4.6.2 Conditions

Supposing the name bound by the definition to be converted is called `foo`, the condition for demoting `foo` from a `where` to a `let` is as follows:

- If there is another definition in the scope of the `where` clause that depends on `foo`: demoting `foo` to a `let` expression will result in that `foo` being passed as a formal parameter (and to all uses of the definition as an actual parameter). Of course this would fail if there are other uses within the `where` clause bound variables rather than in the body of the expression itself.

4.6.3 Design Issues

Supposing the name bound by the definition to be converted is called `foo`, the design issues for demoting `foo` from a `where` to a `let` are as follows:

- Since `foo` will always be added to the outermost scope in the `let` expression, any definitions with the same name in a `let` expression will still have the same behaviour as before as a `let` always binds more tightly than a `where` clause. For example, suppose we select `foo` for demotion to a `let` expression:

  ```
  g x = let foo = 45 in foo where foo = 26
  ```

  As there are two `foo` definitions, the `foo` defined immediately in the `let` expression always binds more tightly, and is therefore always called by the evaluator when evaluating the right-hand-side of `g`. Demoting the `foo` in the `where` clause, therefore, moves `foo` to a `let` expression outside of the scope of the immediate `let` expression already in place:

  ```
  g x = let foo = 26 in (let foo = 45 in foo)
  ```

  The meaning of `g` therefore remains unchanged.

- `foo` will always be added at the top-most level of the list of bindings in the `let` clause of the function body immediately attached to the (defining) `where` clause. If no `let` expression exists, one is created.

- If the body of the function defining the `where` clause of `foo` contains guards, the guards must be converted to an `if..then...else` before the demotion can continue. This condition is explained in further detail on Page 79.

4.7 Case Analysis Simplification

Case Analysis Simplification transforms the case expression into the always succeeding branch, replacing variables that are bound in the case analysis with variables bound outside the case analysis via symbolic evaluation. This is particularly useful if one introduces new pattern matches and then inlines calls on the right hand side.

4.7.1 Example

As an example, consider the following equation:

```
f xs@[] = (case xs of
                [] -> error "Error: empty list"
                (n:ns) -> n) + head (tail xs)
```

Choosing the *Simplify Case Expression* refactoring, we get the following:

```
f xs@[] = error "Error: empty list"
```

where the second branch of the `case` expression has been eliminated.

For an example where `case` analysis simplification might be useful, consider the following definition:

```
f :: [Int] -> Int
f xs = head xs + head (tail xs)
```

By using *Introduction of pattern matches* as described in Section 5.6, we may firstly introduce new equations for `f`:

```
f :: [Int] -> Int
f xs@[] = head xs + head (tail xs)
f xs@(y:ys) = head xs + head (tail xs)
```

In the above, the original equation for `f` has been retained. The main reason for this is so the refactoring does not have to perform any analysis to determine whether or not there is an exhaustive set of patterns defined. Removing the equation automatically may produce a program that the user did not expect.

Now, we unfold the calls to `head` to introduce a `case` analysis:

```
f :: [Int] -> Int
f xs@[] = (case xs of
               [] -> error "Error: empty list"
               (n:ns) -> n) + head (tail xs)
f xs@(y:ys) = (case xs of
               [] -> error "Error: empty list"
               (n:ns) -> n) + head (tail xs)
```

It is obvious that in the first equation, the second branch of the `case` analysis will never be reached, as `xs` is defined as `[]` in the pattern binding. In the second equation, it is also obvious that the first branch of the `case` analysis will never be reached, as the shape of `xs` is not `[]` in its pattern binding; the variable `n` in the second branch is also symbolically identical to `y` defined in the pattern binding of `f`. Simplifying these `case` expressions in turn produces the following code:

```
f :: [Int] -> Int
f xs@[] = (error "Error: empty list") + head (tail xs)
f xs@(y:ys) =  y + head (tail xs)
```

From here onwards, the equations can be simplified further using the refactorings implemented in HaRe —eliminating the call to `head` and `tail`— if we introduce new sub-patterns (defined in Section 5.6.2) for `ys` and inline and simplify the right hand side further. To demonstrate this, consider the equation `f xs@(y:ys) = y + head (tail xs)` as an example:

```
f xs@(y:ys) =  y + head (tail xs)
```

$\rightsquigarrow$ (*Introduce new pattern matching for* `ys`)

```
f xs@(y:ys@(z:zs)) =  y + head (tail xs)
```

$\rightsquigarrow$ (*unfold* `tail xs`)

```
f xs@(y:ys@(z:zs)) =  y + head (case xs of
                                 [] -> error "Error: empty list!"
                                 (a:as) -> as )
```

$\rightsquigarrow$ (*simplify* `case` *statement*)

```
f xs@(y:ys@(z:zs)) =  y + head  ys
```

$\rightsquigarrow$ (*unfold* `head ys`)

```
f xs@(y:ys@(z:zs)) =  y + (case zs of
                                [] -> error "Error: empty list!"
                                (a:as) -> a)
```

$\rightsquigarrow$ (*simplify* `case` *statement*)

```
f  xs@(y:ys@(z:zs)) =  y + z
```

The expression `head (tail xs)` has therefore been reduced to `z` on the right hand side of `f`, eliminating the calls to `head` and `tail` altogether.

4.7.2 Conditions

Suppose the expression to be simplified is `foo`, the conditions for simplifying `foo` are as follows:

- The right hand side of the definition of `foo` must be a case analysis, otherwise an error occurs. Simplifying an `if...then...else` expression of a guard is not possible as the simplifier may diverge during the process. See Section 4.9 for more details.
- Suppose we have the following equation, where the case analysis on the right hand side is selected for simplification:

```
    f = let x@(y,z) = e in case x of
                             (1,2) - > 1
                             (a,b)  -> a
```

A binding analysis is performed on the expression `e` so that, if the second branch of the case analysis succeeds, `a` can be symbolically resolved to either a normal form, or the variable `y`. For example, suppose that `e = (1,2)`, then it is trivial to equate `a` with the value `1`. However, if `e = diverge` (where `diverge` is the diverging term) then evaluating `e` will obviously make the simplifier diverge also. Therefore if `e` is a function call of any kind then it is not evaluated. This issue is explained in more detail in Section 4.9.

4.7.3 Design Issues

Suppose the expression to be simplified is `foo`, the design issue for simplifying `foo` is as follows:

- If any guards occur on the right hand side of the equation then no attempt is made to evaluate them; they are converted to an `if..then..else`.

4.8 Bound Variables

This section discusses an issue that arises due to folding (both simple folding, as described in Section 4.1, and generative folding, as described in Section 4.2). Any matching of expressions with bound variables is up to alpha equivalence of those bound variables. For example, if we are folding against a definition of `foo` that contains any pattern matching on its right-hand-side in the form of a case expression, lambda abstraction or `do` notation, then the structure of the clause must match the structure of any potential substitution instances within the program. The variables in the pattern are renamed so that the two compared pattern clauses contain the same binding information. For example, let us consider that we are folding the following definition of `foo` (if the fold is a *generative* fold, then the following equation of `foo` would be preserved in a comment):

```
foo xs = map (\y -> case y of
                      0 -> 1
                      x  -> x + 1) xs
```

against the following sub-expression:

```
map (\x -> case x of
             0 -> 1
             a  -> a + 1) xs
```

The variables bound in the instance above are renamed to their corresponding binding names in `foo`. The two ASTs can then be compared. In the example above, the variable bound in the lambda abstraction can be renamed to `y`, and the `a` bound in the case analysis can be be renamed to `x`. This is obviously identical to the right-hand-side of the selected equation, `foo`. If any free variables occur in either of the expressions that are being compared, then they are identical if and only if they are the same identifier.

4.9 Symbolic Evaluation

Generative folding (described in Section 4.2) and case analysis simplification (described in Section 4.7) both use the symbolic evaluation techniques implemented and outlined in this section. The symbolic evaluation implemented for the refactorings in this thesis is not supposed to be an attempt at a complete symbolic evaluator, instead it provides the necessary functionality required for the refactorings.

As described by the report [93], Haskell 98 allows pattern bindings to be embedded within expressions; pattern bindings at the expression level occur within `case` expressions, lambda abstractions and `do` notation (including list comprehensions). It is possible, therefore, to bind new patterns to pattern variables that

```
f x@(z:zs) = case x of
                [] -> error "Error: empty list!"
                (y:ys) -> ys
```

Figure 19: A simple `case` expression

have already been defined; in Figure 19, we have a pattern binding on the left-hand-side of `f` defined as an as-pattern `x`. The case expression simply takes `x` and performs further pattern matching on it: the first branch of the case compares `x` with `[]` and the second simply repeats the pattern matching on the left-hand-side of the equation for `f`. This repetition can be removed by the simplifier refactoring as described in Section 4.7, and its underlying implementation as described next.

4.9.1 Case Expression Elimination

Considering Figure 19, it is possible to remove the first branch of the case expression entirely; indeed, it is even better to replace the whole case expression with the second branch (replacing variables bound in the case to variables bound outside).

Although it may be obvious to reader that the branch

```
[] -> error "Error: empty list!"
```

will never be reached, this is determined by evaluating a form of the case expression using the GHC API. First, a closure is created that captures the environment of the case expression, and replaces each branch with an integer value:

```
closure1 x@(z:zs) = case x of
                      [] -> 1
                      (y:ys) -> 2
```

The Haskell project under scrutiny is then loaded into the API, and the above closure is evaluated, passing in calls to `error:error` as formal parameters to `closure1`. The evaluation then returns either *1* or *2* depending on which branch succeeds (in the above case, *2* will always be returned). If an error occurs, or the evaluation fails, a negative integer value is returned which is then processed by the refactorer accordingly.

The refactoring can then determine the AST for the branch by a simple count and then perform symbolic evaluation where necessary.

4.9.2 Symbolic Evaluation Infrastructure

The symbolic evaluator takes a case alternative, a list of patterns (the environment for the case expression) and the expression under the case scrutiny. With this information, the symbolic evaluator performs an analysis over the case expression, and tries to match it with any patterns bound in the environment. If they are matched, they are paired with the patterns bound in the case alternative and the variables in the expression in the alternative are transformed into variables bound in the environment. This new expression is then returned by the symbolic evaluator.

To put this into context, consider, again, the example in Figure 19. After the simplifier (Section 4.9.1) has been applied to the case expression, we are left with the second alternative: `(y:ys) -> ys`. This is passed to the symbolic evaluator together with the environment `x@(z:zs)` and the case expression under scrutiny, `x`. The evaluator associates the `x` in the expression with the as-pattern in the environment. The patterns in the as-pattern are then paired with the patterns in the alternative: `[(z,y),(:,:), (zs, ys)]`; and the expression `xs` in the alternative is re-written to produce `ys`, which is then returned by the symbolic evaluator.

4.9.3 Where and Let Clauses

The environment as described in the section above may also contain patterns defined at the equation level with `where` or `let` clauses. For example, consider

```
f  = let x@[a,b,c] = [1,2,3] in
         case x of
              [] -> error "Error: empty list!"
              [n,m,o] -> [n,m]
```

Figure 20: A `case` expression demonstrating a simple `let` binding

```
f  = let x@[a,b,c] = make_list a in
         case x of
              [] -> error "Error: empty list!"
              [n,m,o] -> [n,m]
make_list n = n : make_list (n+1)
```

Figure 21: A `case` expression demonstrating an infinite `let` binding

Figure 20, where instead of a pattern binding on the left-hand-side of `f`, we now have a `let` clause instead. In this case, the symbolic evaluator infers that the expression `[n,m]` in the `case` alternative is actually `[1,2]` as defined by the `let` expression.

Currently, however, HaRe only looks at `let` or `where` clauses that are one-level deep and do not contain function calls; in this case only the pattern binding on the left-hand-side of the `let` or `where` declaration will be used for the environment, and the right-hand-side will be completely discarded. This is to avoid a complex analysis that may end up in trying to evaluate diverging functions. Consider the example in Figure 21; here, the `let` expression is a call to a diverging term and therefore the pattern binding for `x` in the `let` expression cannot be resolved. In this example, the case alternative `[n,m]` can only be symbolically linked to `[a,b]`.

4.10 Summary

This chapter presented a number of *structural* refactorings that have been implemented in the Haskell refactorer, HaRe. With an established refactoring framework [67] it was possible to choose those refactorings that were more challenging and allowed the HaRe framework to be extended further. The refactorings in this chapter also demonstrated a case study into language design and implementation. To show how these refactorings can be used on a real world example, Chapter 7 shows a case study using these refactorings (and the other refactorings that form this thesis) on a real-world program.

Chapter 5

Data Type Based Refactorings

This chapter describes the new data type based refactorings that have been defined and implemented in HaRe. Following on from the refactoring work by Li [67], and using the refactoring catalogue [96] maintained by Thompson as a basis, a new set of refactorings which aim to complement and extend the existing refactorings in HaRe have been provided. This chapter follows the guide outlined on Page 76 in Chapter 4. The following refactorings are described in this chapter:

- Adding a constructor (Section 5.2);
- Removing a constructor (Section 5.3);
- Adding and removing a data type field (Section 5.4);
- Introducing pattern matching (Section 5.6);
- Introducing pattern matching for a sub-field (Section 5.6.2);
- Introducing a case expression (Section 5.6.3).

The chapter begins with a section regarding the type checking issues that were addressed with implementing the refactorings in this chapter, and concludes with a summary, on Page 128 in Section 5.7.

5.1 Type Checking

Various refactorings in this chapter use the type-checker from GHC to infer the types of sub-patterns.

- The type of the sub-pattern is inferred using GHC as follows: the parameter position of the equation in which the sub-pattern occurs is calculated. For example, if `ys` is selected within:

  ```
  f :: a -> [Int] -> Int
  f x (y:ys) = e
  ```

  `ys` occurs within `f`'s second parameter, therefore the type of the second parameter is then extracted (in this case `[Int]`) from the type signature and a new closure is passed to the GHC type checker: `closure ((y:ys)::[Int]) = ys`. GHC then infers that the type of `closure` is `[Int] -> [Int]`, and it is relatively straight forward to extract the return type from that result. It is important to note that there must be a type signature defined for the function that we want to infer. As of yet, there is no means, via the GHC API, to give it a function and have an inferred type returned. To get round this, by adding a type signature and then passing in a type annotation, in this fashion, it is possible to use the GHC API to get a type of a given expression.

- As a new equation is built with a type annotation to the GHC type checker, any polymorphic types within the type annotation must be removed. Type variables are not allowed to occur within a type annotation due to the scoping of the variables. Consider: `closure ((y:ys)::[a]) = ys`; the type `[a]` is only in scope within the pattern binding `(y:ys)` and not on the right-hand-side of the equation. This will actually give an error within GHC. The refactoring gets around this by replacing all type variables in the annotation with `Int`; it is reasonable to do this, as the refactoring errors if the type of the pattern is of type `Int` anyway (we cannot produce pattern matching over a built in type such as `Int`, as firstly the number of elements is potentially infinite, and secondly, the elements are not explicitly defined in the implementation). The error message is generalised to include pattern match introduction over polymorphic types.

5.2 Adding a Constructor

Adding a constructor to a defined data type. The introduced constructor is added immediately after a selected constructor definition in a data type.

5.2.1 Example

An example of adding a constructor `Var` to a data type `Expr` is shown in Figure 22. In the example, we select the constructor '`Minus`' and choose to add a new constructor immediately after (the result is shown in the right column). We add the new constructor '`Var`' with a parameter '`Int`'. The function '`eval`' is updated automatically to include pattern matching for the newly added constructor.

5.2.2 Conditions

- A condition is that `Foo` is not already a constructor of a type already in scope, if it is, the refactoring terminates with an error.

5.2.3 Design Rationale

Suppose the Constructor to be added is called `Foo` and the data type for `Foo` is called `T`, then the effects of adding a constructor are as follows:

- `Foo` together with the types of the fields or arguments to the constructor are defined by the user and added to the definition of `T`. `Foo` is always added immediately after a user-selected constructor.

Before:

```
data Expr = Plus Expr Expr | Minus Expr Expr

eval :: Expr -> Int
eval (Plus e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
```

After:

```
data Expr = Plus Expr Expr | Minus Expr Expr | Var Int

addedVar = error "added Var Int to Expr"
eval :: Expr -> Int
eval (Plus e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Var a) = addedVar
```

Figure 22: Adding a constructor

- New equations are added to any definition over `T` to allow for pattern matching definitions to include the new constructor. This includes equations in all modules in the current project that are defined over `T`. An exhaustive set of patterns is introduced for `Foo` in all argument positions that have type `T` if the pattern sets are not already exhaustive. The refactoring always preserves the ordering of the pattern matching. If a function, `g`, has an exhaustive list of patterns for `T` already defined, the new pattern match for `Foo` will be placed in the same position as it is placed in the definition of the type `T`. If the patterns in `g` are not in the same order as in the definition of the type of `T` then the new equation is added to the end of the definition of `g`.

- If there is already a pattern match over a type `S` with n cases, plus a variable over `T`, the refactoring adds n new equations, one for each case of `S`. For example, consider we have the following data types, `S` and `T` and a function `foo`:

```
data S = C1 | C2 | C3
data T = T1 | T2
foo :: S - > T -> T
foo C1 x = x
foo C2 x = x
foo C3 x = x
```

Adding the constructor `T3` to `T` has the effect of adding a new pattern match for each clause already defined for `S` in `foo`:

```
data S = C1 | C2 | C3
data T = T1 | T2 | T3
addedT3 = error "added T3 to T"
foo :: S - > T ->T
foo C1 T3 = addedT3
foo C1 x = x
foo C2 T3 = addedT3
foo C2 x = x
foo C3 T3 = addedT3
foo C3 x = x
```

If the pattern matching contains a variable in all defining locations, the new pattern match for `Foo` is placed immediately before the first equation.

- It is possible that the user gives general type variables as parameters to `Foo` that do not occur on the left-hand-side of the data type implementation. For example, if the definition for the data type is: `T a = C1 a` and `Foo a b` is added as an additional constructor with `a` and `b` as type parameters. If this is the case, the variables must be added to the list of the data type parameters (to produce `T a b`) and all type signatures must be updated to reflect the addition of the new type parameters. If the type parameter is used in the type signature under refactoring, a new polymorphic type variable is introduced, so long as the name does not conflict with any other type variables in scope.

5.2.4 Design Issues

This refactoring uses the type checking facilities of GHC (as described in Section 5.1). When adding a constructor, it is possible that the refactoring needs to be able to check the type of pattern variables, particularly those occurring in a nested pattern match. Rather than performing a rudimentary type checking analysis using the type signatures of the definitions (which may not always be present) under refactoring, it seemed far easier to build a mechanism to allow GHC to return a type for a particular pattern variable of interest. Consider, for example:

```
data T a = T1 a  | T2
data S = S1

f :: (Int, T S) -> Int
f (x, T1 y) = 42
```

Supposing a new constructor, `S2`, is added to the definition of `S`. Using the behaviour specified in Section 5.2.3 means that a new equation must be introduced for `f`:

```
data T a = T1 a | T2
data S = S1 | S2

addedS2 = error "Added constructor S2 to S"
f :: (Int, T S) -> Int
f (x, T1 S2) = addedS2
f (x, T1 y) = 42
```

GHC is used to infer the type parameter for the pattern match against `T1`; this is so type checking analyses do not have to be implemented into the refactoring.

5.3 Removing a Constructor

Removing a constructor is defined as the inverse of adding a constructor and is not a refactoring but rather a destructive transformation in that it potentially eliminates equations (and therefore does not preserve the program's behaviour). Removing a constructor allows a constructor to be identified and all equations defined over the constructor as a pattern match are commented out. All occurrences of the constructor in an expression are replaced with calls to `error`.

5.3.1 Example

An example of removing a constructor `Var` from a data type `Expr` is defined in Figure 23 on Page 117 (left to right). `Var` is selected for removal and the refactoring removes the value from its defining definition, `Expr` and comments out all equations referring to the value `Var` in a pattern. When used on the right hand side `Var` is replaced with a call to `error`.

5.3.2 Design Issues

Removing a constructor called `Foo` in a destructive manner from a data type `T` has the following implications:

- `Foo` is removed from the data type implementation of `T`. Any redundant type parameters (i.e. parameters no longer used on the right hand side of the definition) left as a result are also removed. For example, if the data type is defined: `T a b = C1 a | Foo a b` and `Foo a b` is removed, the updated type for `T` will be: `T a = C1 a`. All type signatures are also updated to reflect the change in type implementation. Removing redundant type parameters could also be implemented as a stand-a-lone refactoring, and the stand-a-lone implementation is currently left as future work.

Removing redundant type parameters in this fashion may, in some rare instances, result in programs that no longer type check. Removing redundant parameters is not behaviour preserving and can be used by the programmer occasionally to facilitate type unification. This detail is discussed in more detail in Section 5.5 on Page 120.

- Any equations with a pattern match using `Foo` are commented out. This is so that equations can still be referenced in the program source code at a later date if the programmer wishes.
- Any references to `Foo` occuring within an expression are replaced with calls to `error`. The error message informs the user that the constructor was removed and also gives the function name that defines the expression.
- Any references to a definition that has been commented out as a result of the refactoring are replaced with calls to error.
- If there are any pattern bindings involving `Foo` in the program then the refactoring terminates and gives an error. Otherwise, the refactoring would also have to eliminate all uses of variables bound within the erroneous pattern binding. For example, `Foo x y = e` is defined as a pattern binding within an expression, all uses of `x` and `y` must also be eliminated in that scope. However, it is possible that if `Foo` occurs within a pattern binding of the form `(Foo x y, Blah z w) = ...` then it may be possible to salvage the pattern match for `Blah`.

This refactoring may be useful if the programmer defines a constructor that needs to be modified: removing the constructor destructively and then adding a new constructor allows them easily to correct this mistake. It is understood that this sort of destructive behaviour for a refactoring tool may be dangerous: eliminating equations and expressions in this manner changes the meaning of a program.

Before:

```
data Expr = Plus Expr Expr | Minus Expr Expr | Var Int

addedVar = error "added Var Int to Expr"
eval :: Expr -> Int
eval (Plus e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Var a) = addedVar
```

After:

```
data Expr = Plus Expr Expr | Minus Expr Expr

addedVar = error "added Var Int to Expr"
eval :: Expr -> Int
eval (Plus e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
{- eval (Var a) = addedVar -}
```

Figure 23: Destructively removing a constructor

5.4 Adding a Field to a Constructor of a Data Type

Adding a field to a constructor allows a new field to be added to an identified data type constructor.

5.4.1 Example

Figure 24, from top to bottom, shows an example of a new field being added to a data type. The new field, of the polymorphic type `b`, generalises the data type further. `b` is added to the left hand side of the type definition, and also to all corresponding type signatures of `Data1` in the program.

5.4.2 Conditions

Adding a field `t` to a constructor `Foo` has the following implications:

- The user inputs a type for the new field. If the type of the parameter is a new polymorphic type not already defined on the left-hand-side of the data type, then the parameter is added to the left-hand-side of the data type implementation and all type signatures are updated accordingly. `undefined` is used as the value for the new field in cases where the constructor is used.
- If the data type is defined as a record type, the user can input a field name as well as a type. The name must be unique to `Foo` otherwise the refactoring gives an error and terminates.

5.4.3 Design Issues

The design issues of adding a field `t` to a constructor `Foo` are as follows:

- All uses of `Foo` within the program are updated to take into account of the new field, this includes pattern matching and variable binding.

- `t` is always added to the beginning of every occurrence of `Foo` in the program so that partial applications can be dealt with correctly. The reason for this was to simplify the implementation of the refactoring. Rather than perform an analysis into all cases where a partial application may be presented and producing a workaround, it is far easier to always add the field as the first parameter of the constructor. Indeed, *adding a parameter* [67] has been defined to behave in exactly the same way for the same reason.

- There might be cases where the user wants to add a field to all the constructors of the data type. It is possible to do this by adding a field to each of the constructors of the data type in turn.

5.4.4 Comments

One of the major drawbacks of functional programming in general is that if one adds a field to a data type, a substantial amount of code modification has to take place in order to introduce new pattern matching for all functions defined over that data type. This is not a problem in object-oriented programming languages where data types are represented by classes. Adding a new field to a class is implemented by adding a new subclass to the class hierarchy; this has an advantage that the methods do not need to be updated to take into account the newly added subclass.

5.5 Removing a Field

Removing a field is the complement of adding a field as described in Section 5.4.

5.5.1 Example

Figure 24 on Page 123, from *bottom* to *top*, shows an example of a field being destructively removed from a data type. The field in question, of the polymorphic type `b`, is removed from the left hand side of the type definition, and also from all type signatures involving `Data1`.

5.5.2 Design Issues

The implications of removing a field `t` from the constructor `Foo` *destructively* are as follows:

- The field `t` can not in general be identified by its type as there may be cases where a constructor has two or more fields of the same type. Instead, the field is identified by its position in the constructor application.
- Any variables referring to `t` in a pattern binding are removed from the binding; any occurrences of the variables are then replaced with a call to `error` in that scope. The error message informs the user where the binding for the field has been removed.
- References to `t` in a constructor application within an expression are simply eliminated from the program.
- The list of type parameters for the data type is updated to take into account any redundant parameters after `t` has been removed. All type signatures are updated to reflect this change.

  It is important to note that whilst removing redundant type parameters is not behaviour preserving, it may conflict with programs that rely on type parameters to resolve type variable unification. For instance consider the following Haskell program:

  ```
  data Vector a = Vector a

  data PackType a = Pack a  deriving Show

  class Pack a where
  ```

```
  pack :: Vector a -> PackType a

instance Pack Int where
  pack (Vector a) = Pack a

instance Pack Char where
  pack (Vector a) = Pack a

test :: Pack a => Vector a -> PackType a
test v = pack v

dummy = test (Vector ’c’)
```

Suppose we remove the field `a` from the constructor `Vector`:

```
data Vector = Vector

data PackType a = Pack a  deriving Show

class Pack a where
  pack :: Vector -> PackType a

instance Pack Int where
  pack (Vector) = Pack undefined

instance Pack Char where
  pack (Vector) = Pack undefined

test :: Pack a => Vector -> PackType a
test v = pack v
```

```
-- dummy :: PackType Int
dummy = test (Vector {-'c'-})
```

This now fails to type check as it is unclear which overloaded version of `pack` to use. The type variable in `Vector` was being used to determine the type of `PackType a` and consequently the correct version of `pack` to call. Although this is an extreme example, it is important to note that unification in this way can be used to determine which overloaded version of an operator to use.

In contrast the *safe* variant is defined as follows:

- `t` is selected by the user and an error is given if the cursor has selected an erroneous entity. This is a user interface issue, it is easy to accidently select an entity in the program that is not a constructor field.
- If there is no reference to `t` in a pattern binding —either occurring within a function match or an explicit pattern binding— or within a constructor application within an expression, the field is safely removed.
- The list of type parameters for the data type are updated to take into account any redundant parameters after `t` has been removed from `Foo`. All type signatures are updated to reflect this change.

5.6 Introducing Pattern Matches Over an Argument Position

This refactoring has three flavours: introduction of pattern matches for a top-level variable on the left hand side of an equation, introduction of pattern matches for a sub-pattern variable occurring on the left hand side of an equation, and introduction of a case analysis for top-level variables and sub-level variables.

Before:

```
data Data1 a = C1 a Int Char |
               C2 Int        |
               C3 Float

f :: Data1 a -> Int
f (C1 a b c) = b
f (C2 a) = a
f (C3 a) = 42

g (C1 (C1 x y z) b c) = y

h :: Data1 a
h = C2 42
```

After:

```
data Data1 b a = C1 a Int Char | C2 Int | C3 b Float

f :: (Data1 b a) -> Int
f (C1 a b c) = b
f (C2 a) = a
f (C3 c3_1 a) = 42

g (C1 (C1 x y z) b c) = y

h :: Data1 b a
h = C2 42
```

Figure 24: Adding and removing a field

5.6.1 Introduction of Pattern Matches

Introducing pattern matches for a function with a variable in a particular argument position in all its defining equations, replaces the variable by an exhaustive set of patterns over the type of the variable.

Conditions

Supposing new patterns are to be introduced for a pattern `x` (within the equation `f x = head x + head (tail x)`), the conditions for introducing new patterns for `x` are as follows:

- The definition of `f` must have a type signature defined. The type signature is used to calculate the type of `x`. The new matches are then calculated by traversing the list of in-scopes (of identifiers) within the project until the type is found, together with its constructors and constructor arities. Currently, the GHC API does not have a method for inferring the type of a given function, which is why a type signature must be defined; this issue is discussed in more detail on Page 110, in Section 5.1.
- The type of `x` must not be completely polymorphic, i.e. a type variable. Type variables do not have concrete values defined and hence no new matches can be defined. The restriction comes back to the work around for GHC's API that is defined on Page 110.
- The type of `x` must not be a built in Haskell type such as `Int`. Built-in types do not have defined constructors, but instead, rely on pre-determined values that are usually built into the compiler implementation. The in-scope analysis determines whether a type has any constructors defined for it or not, built-in types (such as numerical types like `Int` and `Float`) do not register as having constructors defined in that context.
- The selected pattern must be a variable in all defining locations. The refactoring always tries to introduce a complete set of exhaustive patterns for all

arguments; having the pattern as a variable makes this analysis much more straight-forward to realise.

- `x` must be a pattern binding occurring on the left-hand-side of an equation and not defined within an expression.

Design Issues

The implications of introducing new pattern matches for a pattern variable are as follows:

- The new pattern matches are added to the definition of `f` and the introduced patterns for `x` are placed within an as pattern. If the equation for `f` and its type signature are as follows:

  ```
  f :: [Int] -> Int
  f x = head x + head (tail x)
  ```

  Then the new pattern matches for `f` will be as follows:

  ```
  f :: [Int] -> Int
  f x@[] = head x + head (tail x)
  f x@(y:ys) = head x + head (tail x)
  ```

  The right-hand-side is copied into the new equations and any new pattern variables that are introduced are given new, distinct, names so that no name conflicts will occur.

  It is also possible to take the introduced pattern further, so that the user gets the variable from the head position of the introduced pattern instead of `head x`. An example of this is given on Page 100.

- The original equation is left after introducing the exhaustive pattern match. One reason for this is to cover the `undefined` case, in case the implementation of the type of the arguments to the equation change after the refactoring

Before:

```
f :: Either [Int] Int -> Int
f (Left a) = head a + head (tail a)
f (Right a) = a
```

After:

```
f :: Either [Int] Int -> Int
f (Left a@[]) = head a + head (tail a)
f (Left a@(x:xs)) = head a + head (tail a)
f (Left a) = head a + head (tail a)
```

Figure 25: Introducing pattern matches for a sub pattern

is performed. We leave it to the user to remove the left over equation if they desire.

5.6.2 Introduction of Pattern Matches for Sub-patterns

Introducing sub-pattern matches is in many ways very similar to *introducing pattern matches* as described in 5.6.1. However, in this section we introduce a variation of introducing patterns: the ability to introduce new patterns for a sub-pattern.

Example

An example of introducing new pattern matches for a sub pattern is shown in Figure 25.

Conditions

The conditions for introducing new patterns for `ys` are analogous to those in 5.6.1.

Before:

```
f :: Either [Int] Int -> Int
f (Left a) = head a + head (tail a)
f (Right a) = a
```

After:

```
f :: Either [Int] Int -> Int
f (Left a) = case a of
               x@[] -> head x + head (tail x)
               x@(y:ys) -> head x + head (tail x)
f (Right a) = a
```

Figure 26: Introducing a case analysis for a sub pattern

5.6.3 Introduction of Case Analysis

Introducing a case analysis seemed to be a natural extension to the introduction of pattern matches. Instead of creating new pattern *matches* it is also possible to create a new *case analysis* over a pattern variable. This refactoring works for both sub-patterns and top-level patterns occuring on the left hand side of an equation.

Example

An example of introducing a new case analysis for a pattern is shown in Figure 26. In this example, we select the variable `a` in the first equation for `f` to introduce new pattern matches for `a` as show in the column on the *right*. As `a` has the type `[Int]` new patterns have been introduced to cover all constructors defined in the list type, these are: `[]` and `(:)`.

Conditions

The conditions for this refactoring behave in the same way as described in sections 5.6.1 and 5.6.2.

5.7 Summary

This chapter presented a number of *data type* based refactorings that have been implemented for the Haskell refactorer, HaRe. The refactorings here aim to complement the set of structural refactorings described in Li's thesis [67] and in Chapter 4 wherever possible. For example, it is possible to introduce and remove a *definition* just as it is possible to introduce and remove a *value* of a data type. Adding and removing a parameter to a function is the analogue to adding a removing a field to a data type constructor. Also, just as dead code elimination is described in Chapter 3, dead type variables are also removed from the left hand side of type implementations and type signatures if they are no longer needed after a field or constructor removal has been performed. Chapter 7 shows a real-world example of these refactorings in practice.

Chapter 6

Clone Detection and Removal

6.1 Introduction

Duplicated code or very similar code (often called code clones) is a well known issue in refactoring and software maintenance. The term duplicated code, in general, refers to a code fragment that is identical or similar to another code fragment; the exact meaning of "similar", however, varies between different programming contexts.

Several studies have shown that software systems with code clones are more difficult to maintain than the software systems with little or no duplicated code [51, 6]. It is therefore beneficial to remove clones quickly after their introduction by the use of refactoring technology.

Software clones appear for a variety of reasons; the most obvious of which is the use of the copy and paste function in an editor. Clones introduced by this mechanism often indicates a poor design with a lack of function abstraction or encapsulation [18, 72]. Refactoring can be used to eliminate this problem either by first refactoring the code to make it more reusable, without duplicating code; or by refactoring out the code clones at a later stage [32]. In the last decade, a substantial amount of research has gone into duplicate code detection and removal from software systems; the sheer size of the recent survey on the topic of clone detection is well over 100 pages [101]. However, few such tools are available for functional programs, and there is a particular lack of clone detection support within existing program environments.

Common terminology for the clone relations between two or more code fragments are the phrases *clone pair* and *clone class* [55]. A clone pair is a pair of code fragments that are identical or similar to each other; a clone class is the set of code fragments which are identical or similar to each other within a code base.

The rest of this chapter details a clone detection technique for Haskell built into the framework of HaRe. The tool should be able to handle large Haskell programs and report as many clones as possible. Fully automating the tool would be dangerous because it could lead to results that the user did not expect; it is better to give the user control over which clones should be removed and extracted. The clone detection stage is intended to be fully automatic, while the transformation stage is not. Being part of a programming environment and a large-scale refactoring tool, the clone detection tool is far more likely to be useful to working Haskell programmers.

The clone detection technique presented in this chapter has a stronger criterion than other clone detection mechanisms: it is concerned with finding clones which are common substitution instances of expressions. The clone detection technique is able to report code fragments in a Haskell program that are syntactically identical after an alpha variable renaming. Syntactically, these code clones are a sequence of well-formed expressions or functions, and therefore this approach makes use of the token stream and AST provided in HaRe as part of Programatica. Unlike a fully automated removal of clones, which usually does not give appropriate solutions in all cases, a different approach is taken here which respects the importance of user intervention during the clone removal process. This intervention allows clones to be removed in a step-by-step process under the programmer's control, and allows for the preservation of clones of particular kinds (for example very small clones). In addition to the above, the clone detection technique has been extended to look for repeated sequences of monadic "commands". It is particularly interesting that clone analysis in Haskell allows us to do this, and even more important is that it is a real contribution to the field of clone detection and elimination.

Section 6.2 gives an overview of work relating to duplicated code elimination. Section 6.3 discusses the analysis stage of the clone removal process for HaRe and Section 6.4 discusses the transformation stage of the clone removal process.

6.2 Related Work

A code clone, in general, refers to a code fragment that has identical or similar code fragments to it in a program's source code. Clone fragments often occur through the use of copying techniques by the developer (usually via cut and paste) with or without minor modifications in a system. If there are no modifications, or the modifications are only a change to the copied fragment, then the original and copied fragments are called code clones and form a clone pair. Two code fragments can be identified as being similar if their program texts are similar, or their semantics are similar (without necessarily being textually similar). Due to the undecidability of function equality [58], the research reported in this thesis only considers syntactically identical or similar code fragments, which can be compared on the basis of their syntactic or internal representation.

This thesis identifies clones that are identical only after a consistent renaming of local variables; top-level identifiers are not renamed (the clone analysis only looks for cut-and-paste instances). The process here obviously ignores variations in literals, layout and comments. The work presented here complements that of Li on clone detection in Erlang [70]. The difference is that Li first compares clones using an analysis of the token stream, only using the more costly AST once clone candidates have been identified. Li also compares clones that are identical after renaming all function names and variables names to the same name respectively.

Typically, the clone recognition process transforms the code into an intermediate representation and then performs a comparison over this representation. A recent survey of existing techniques is given by Roy and Cordy in [101], an overview of which is given in the remainder of this section.

6.2.1 Text-based Techniques

There are several clone detection techniques that are based on pure text-based/ string-based methods. In this approach, the target source program is considered as a sequence of lines/strings. Two code fragments are compared with each other to find sequences of same text/strings. Once two or more code fragments are found to be similar in their maximum possible extent (e.g. with respect to maximum number of lines) they are returned as clone pairs or clone classes by the detection technique.

Dup by Baker [6] uses a sequence of lines as a representation of source code and detects clones. Therefore, it uses a lexer and a string matching algorithm on the tokens of the individual lines. Dup removes all tabs, whitespace and comments from the program. Dup also replaces identifiers of functions, variables and types with a special parameter and concatenates all lines to be analysed into a single text line. Dup then uses a suffix tree algorithm to extract a set of pairs of longest matches. Dup can also generate scatter-plots of found matches. Detection accuracy in Dup is fairly low e.g. it cannot detect code clones written in different coding styles.

Johnson in [51] proposed a pure text-based approach for a redundancy finding mechanism on a substring of the source code. In this algorithm, clone signatures calculated per line are compared in order to identify matched substrings. First, a text-to-text transformation is performed on the considered source file for discarding the whitespace and comments. Following this, the whole text is subdivided into a set of substrings so that every character of the text appears in at least one substring. Matching is then performed to match substrings together, in this stage a further transformation is applied on the raw matches to obtain better results.

6.2.2 Token-based Techniques

Token-based techniques typically first lex/transform the program into a sequence of tokens, and then apply comparison techniques to find duplicated subsequences of tokens; finally, the original code portions representing the duplicated subsequences are returned as clones. Compared to text-based approaches, a token-based approach is usually more effective against code changes and aesthetic structural changes such as formatting and spacing.

CCFinder [55] is one of leading token-based techniques. First, each line of the source file is transformed into a single token sequence; the token sequence is then transformed using language-specific transformation rules. Transforming the language in this way aims at regularisation of identifiers and identification of structures. Each identifier corresponding to a type, variable or constant is replaced with a special token. Replacing identifiers ensures that the algorithm matches different code fragments that are equivalent but have different variable names. CCFinder then uses a suffix-tree based approach to find similar subsequences as clone pairs or clone classes.

Dup [6] is also a token-based clone detector in that it also uses a lexer to tokenize the source code; the tokens of each line are then compared based on suffix-tree analysis. Unlike CCFinder, Dup does not apply the transformation rules. However, Dup does introduce the notion of parameterized matching by a consistent renaming of identifiers.

CP-Miner [72] is used for identifying a similar sequence of tokenized statements. Due to sequential analysis in CCFinder and Dup, the two tools generally don't work well with statement reordering and code insertion. A reordered or inserted statement can break a token sequence that may otherwise be regarded as duplicate to another sequence.

6.2.3 Tree-based Techniques

Tree-based approaches search for similar subtrees in the AST with some tree matching techniques. In the tree-based approach, programs are parsed into an abstract syntax tree (AST). Similar subtrees are then searched for in the tree and the corresponding source code of the similar subtrees is returned as clone pairs or clone classes. The AST contains all the information required to do the clone detection; variable names and literal values are discarded in the tree representation allowing more sophisticated methods for the detection of clones to be applied.

CloneDR [9] is one of the pioneering AST-based clone detection techniques. A parser is used to generate an annotated syntax tree (AST) and compares nodes in its subtrees by characterization metrics based on a hash function through tree matching. The source code of similar sub-trees is returned; CloneDR can check for consistent renaming.

DECKARD [50] is another AST-based language independent clone detection tool; its main algorithm is to compute certain characteristic vectors to approximate structural information within ASTs and then cluster similar vectors, and thus code clones.

There are also clone detection techniques based on program dependency graphs as demonstrated in [59]. Most of the above mentioned clone detection techniques are targeted at large legacy systems, and are not tightly integrated into any kind of programming environment. Language independent clone detection tools tend to have much lower precision since they do not take into account (and cannot) static scoping rules of the particular language under analysis; language independent clone detection techniques are not very suitable for mechanical clone refactoring.

6.3 The HaRe Clone Detector

This section discusses the analysis stage of the clone removal, and Section 6.4 discusses the transformation stage. In this section, Section 6.3.1 discusses AST-level clone analysis and Section 6.3.2 discusses comparing expressions using an AST.

The work presented here has mostly been done independently of the clone detection and removal for Erlang, using Wrangler [70]. The process described for Wrangler uses a token based technique for identifying code clones. The process here is somewhat different, in that the HaRe approach uses a purely AST based approach, reusing the experience gained from designing and implementing refactorings that work over large ASTs.

There are two separate parts of clone detection and removal: an analysis stage, which looks over a Haskell project and calculates clone classes; and a transformation stage, which transforms identified clones from the clone classes in the first stage into calls to an appropriate abstraction.

The HaRe Clone Detector reports clones by identifying duplicated instances of expressions within a Haskell project. Each clone is reported with a start and end location and clones that are instances, or near instances, of the same expression are grouped together to form clone classes. The analysis stage allows the user to specify the minimum token size of the clones. These clone classes are then delivered in a report; see Appendix D for an example of a clone report for the duplicated expressions found in the source code in Appendix C (the source code was provided by Thompson, and first appeared in his text on Haskell [108]).

6.3.1 AST-Level Clone Analysis

The HaRe clone analysis works over an AST representation of a Haskell program, only using the token stream to identify clone classes that have a minimum token size (configured by the user). By using an AST, the analysis detects and transforms clones allowing the clone detector to be easily built upon the existing HaRe framework; source location information is still retained in the AST and white-space and comments are removed.

The analysis is started by the user opening a source file into Emacs or Vim (preferably the Main module) and choosing clone detection from the HaRe drop-down menu. The following describes the stages of the clone analysis:

1. The clone analysis first calculates the export relations of the module (the modules that import the current module either directly or indirectly). When a new project is initiated within HaRe, Programatica stores the export information in a local directory; it is possible to use this information within HaRe to calculate the files exported or imported by the current module under

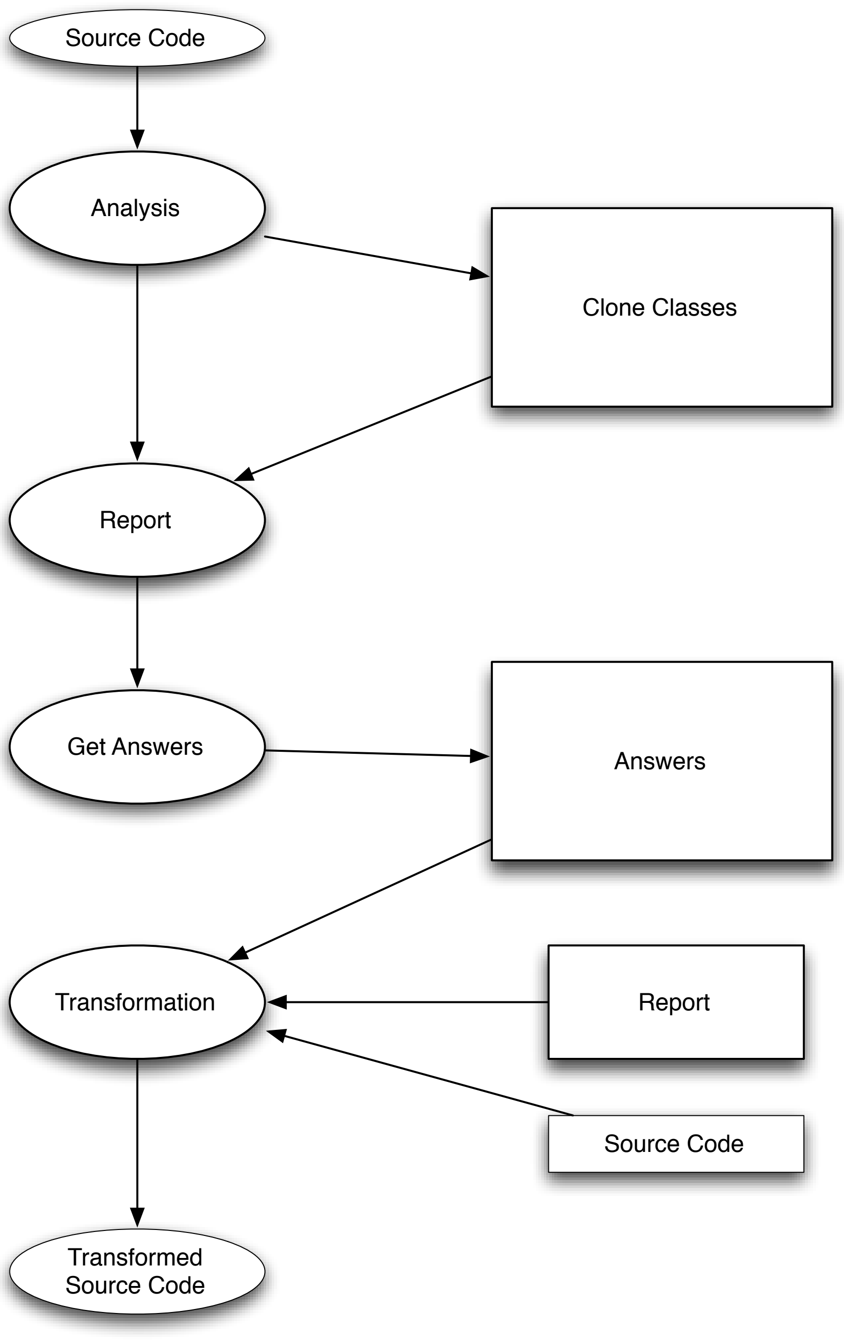


Figure 27: An overview of the clone detection process

refactoring. The export relations are calculated in order for the analysis to determine which modules need to be compared with each other. The clones for a single module project (say module A) are determined by comparing A against itself. If we are comparing a multiple module project that contains the modules A, B and C, we need to be able to compare each different pair of modules, that is n(n + 1)/2 comparisons for an n module project.

2. An initial pass over the AST groups together all expressions of the same shape. Originally, before this grouping, the clone analysis took approximately 15 minutes to compute clones for the Huffman example. The grouping significantly diminished this time down to 50 seconds. By grouping expressions in this way we cut out a lot of irrelevant analysis (i.e. by comparing an application with a section, say). For example, all function applications, identifiers, infix applications, and sections, etc. are grouped together. The grouping also takes into account sub-expressions, so that the expression,

```
(convert (cd++[L]) t1) ++ (convert (cd++[R]) t2)
```

is grouped into the categories as shown in Table 1.

Parenthesized expression	*Function application*	*Infix operator application*	*List*	*Identifier*
`(convert (cd++[L]) t1)`	`convert (cd++[L]) t1`	`(convert (cd++[L]) t1) ++ (convert (cd++[R]) t2)`	`[L]`	`convert`
`(convert (cd++[R]) t2)`	`convert (cd++[R]) t2`	`cd++[L]`	`[R]`	`cd`
`(cd++[L])`		`cd++[R]`		`L`
`(cd++[R])`				`t1`
				`convert`
				`cd`
				`R`
				`t2`
				`+`
				`+`

Table 1: Expressions grouped for clone analysis

3. For each expression in the module, the type of the expression is located and the relevant group of expressions is traversed. A disadvantage is that some clones are missed as the clone analysis is only able to compare expressions with a similar shape. It is not currently possible to compare infix expressions with function applications for example (and those would be not be clones in the sense that we have chosen to work with in this thesis). However, the clone analysis does detect clones between expressions and parenthesized expressions, i.e. `e` and `(e)`.

4. The clones that are found are sorted and grouped together; clones that are contained within larger clones are removed.

5. The clones are pretty printed to a report file. The report file includes the clone classes and their location and module information.

6.3.2 Abstract Syntax Tree Comparison

Two expressions are compared by traversing their structures. If the two structures contain different top-level identifiers at any point during the comparison then the analysis terminates and another expression is chosen by the traversal strategy.

- Suppose we are comparing two identifiers `i1` and `i2`. `i1` and `i2` are identical if either of the following hold:
  - `i1` and `i2` are both locally defined identifiers. If they are locally defined, they may be passed in as formal parameters to an abstraction in the transformation stage.
  - `i1` and `i2` both refer to the same top-level identifier. If `i1` and `i2` refer to different top-level identifiers, then they are not duplicate expressions. This eliminates cases such as comparing `1 + 2` with `1 - 2` (`(+)` and `(-)` are different top level identifiers). If `i1` and `i2` are both local variables then they will always match. A literal may be compared with

any other literal or a locally defined identifier, as literals can be passed in as parameters.

- Local identifiers may be compared with each other as local identifiers can be passed in as parameters.

- The analysis allows for the comparison of expressions with parentheses with expressions without parentheses. For example, suppose we are comparing `(e1)` with `e2`: `(e1)` and `e2` are duplicates if `e1` compares with `e2` structurally, or either `e1` or `e2` are identifiers or literals. If the expression is being compared against a literal of a locally defined variable, then the expression can be passed in as a parameter.

- For expressions containing patterns, for example lambda abstractions and case alternatives, the expressions are the same if their pattern structures are also the same and their expressions are the same. For example, suppose we are comparing the two lambda expressions:

  ```
  e1 = \(x:xs) -> head xs
  e2 = \(y:ys) -> head ys
  ```

  If the structure of the patterns in both `e1` and `e2` are the same then we rename the patterns in `e2` to match those in `e1`; identifiers in the expression `head ys` are also renamed to match the renamed patterns. Therefore, `e2` is renamed to:

  ```
  e2 = \(x:xs) -> head xs
  ```

  Which of course is equivalent to `e1` and therefore duplicated. Lambda bindings are treated in example the same way as `let`/`where` bindings (in terms of their binding resolution). Consider an example where two lambda expressions would not match.

```
f1 = \(x:z:xs) -> x + length xs
f2 = \(x:z:xs) -> z + length xs
```

In the above example, the two expressions within the lambda bodies are different. The first lambda expression refers to `x` while the second refers to `z`. The process used here is exactly the same as the technique used for folding, as described in Section 4.1.

- Expressions are compared up to alpha-equivalence.
- The analysis looks for duplicated monadic sequences and duplicated monadic bindings or a combination of the two. For example, we may have the following two monadic blocks:

```
f = do
   x <- return 565; y <- k 56
   putStrLn (y ++ show x)



h = do
    a <- return 1111; b <- k 56
    putStrLn (b ++ show a)
    return 4
```

The clone analysis traverses through the monadic sequences looking for subsequences that are duplicated. In the above, the clone analysis first determines that the two binding structures are identical up to changes in literals (indicated by the highlighting) by renaming the patterns in the second block so that they match the patterns in the first block:

```
f = do
   x <- return 565; y <- k 56
   putStrLn (y ++ show x)
```

```
h = do
      x <- return 1111; y <- k 56
      putStrLn (y ++ show x)
      return 4
```

After this renaming, the analysis then determines that the two expressions `putStrLn (y ++ show x)` and `putStrLn (y ++ show x)` are also duplicates and therefore labels the binding generators `x` and `y` together with the qualifying statement `putStrLn (y ++ show x)` in both blocks as being clones.

This section discussed the analysis stage of clone detection. The analysis stage looks over a multiple-module Haskell project and reports clones in the form of clone classes into a report file. The section preceding this discusses the transformation stage of clone elimination.

6.4 Refactoring Support for Clone Removal

This section discusses the transformation stage of clone elimination. In Section 6.4.1 abstracting clones is discussed; Section 6.4.2 discusses the implementation of a step-by-step clone extractor; Section 6.4.3 discusses introducing the abstraction; Section 6.4.4 shows how the abstraction is calculated and Section 6.4.5 details how monadic sequences and bindings are wrapped in an abstraction.

If a program contains a large amount of duplicated code, then it is a reasonable to assume that the program lacks abstraction and design. One of the purposes of this thesis, therefore, is to implement refactoring technology to remove duplicated code from within a Haskell program. Already discussed in previous chapters are the refactorings *function folding*, *as-pattern folding* and *merging* that —as a side-effect— help to remove duplicated code (and may also, however, help to introduce

```
-- Order on first entry of pairs, with
-- accumulation of the numeric entries when equal first entry.

alphaMerge :: [(Char,Int)] -> [(Char,Int)] -> [(Char,Int)]

alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)

                                              (A)

  | (p==q)          =  (p,n+m) : alphaMerge xs ys

                                              (B)

  | (p<q)          = (p,n) : alphaMerge xs ((q,m):ys)

                                              (C)

  | otherwise          =  (q,m) : alphaMerge ((p,n):xs) ys
```

Figure 28: Duplicated instances of the same expression

duplicated code). These refactorings rely on the user identifying particular problematic areas, and then invoking the refactorings accordingly. The clone detection tool is designed to perform the analysis automatically and to allow the user to choose whether or not to create an *abstraction* for a particular clone class. Furthermore, the *undo* feature of HaRe allows the user to recover from a transformed program if they change their mind.

6.4.1 General Function Abstraction

This subsection introduces a new refactoring for HaRe in the form of function abstraction. This refactoring abstracts expressions within a clone class into a call to a new function. The refactoring takes into account instances of an expression together with exact duplicates of the expression. To show how this abstraction works, consider the code fragments in Figure 28. In this example, the HaRe clone detector has singled out the code fragments as duplicated instances of each other. In each code fragment *(A)*, *(B)* and *(C)* similar calls to the function `(:)` occur. The HaRe clone detector reports these as a clone class to the user, and asks whether or not they would like form an abstraction over this expression. A new function is created automatically with the expression indicated in the clone class as the function body; any formal parameters are added to the function abstraction; for example, if the clone differs in a literal at each instance this is made into a parameter. The expression instances are then replaced with a function call. The following gives an example of the abstraction that is created for the instances highlighted in Figure 28:

```
abs_1 p_1 p_2 p_3 p_4
 = (p_1, p_2) : (alphaMerge p_3 p_4)
```

Figure 29 shows the transformed program after the abstraction is created and the expressions are replaced with calls. This has the advantage of removing duplicated code from within the program and encourages code reuse and maintenance. The programmer now only needs to worry about maintaining one call to `abs_1`, while before there were three calls to `(:)` to maintain. It is interesting to note, however, that in the example above the introduction of the abstraction does not shorten the expressions; it does abstract away a common sub-expression though, allowing the expression to be more easily maintained at a later stage in the program development cycle.

6.4.2 Step-by-step Clone Detection

The clone detection is designed to be fully automatic. Clone classes that are detected are presented to the user in the form of a report, where the user can then step through the instances, deciding whether or not they should be abstracted.

To put this into context, consider the code fragments in Figure 30. The HaRe clone detector has detected two clone classes, the first is represented by the code

(A)

```
   abs_1 p_1 p_2 p_3 p_4
    = (p_1, p_2) : (alphaMerge p_3 p_4)

-- Order on first entry of pairs, with
-- accumulation of the numeric entries when equal first entry.

alphaMerge :: [(Char,Int)] -> [(Char,Int)] -> [(Char,Int)]

alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)
  | (p==q)         = abs_1 p (n + m) xs ys
```

(B)

```
  | (p<q)         = abs_1 p n xs ((q, m) : ys)
```

(C)

```
  | otherwise         = abs_1 q m ((p, n) : xs) ys
```

Figure 29: Duplicated instances replaced with a call to `abs_1`

highlighted in block *(A)*. In code block *(A)*, HaRe has identified the expressions on lines 6 and 9 to be instances of the same expression. HaRe shows this by delivering a report to the user, stating that the sub-expression on line 6 is an instance of the sub-expression on line 9. For each of the instance matches, HaRe asks the user whether or not they would like to replace the sub-expression with a call to an abstraction. The following is an example of the interaction:

```
/home/cmb21/huffman/CodeTable.hs  ((46,5),(46,44)):
                                 >spaces m ++ show c ++
                                  " " ++ show n ++ "\\n"<


Would you like to extract this expression (Y/N)?
```

For each instance that matches, HaRe displays the start and end locations of the clone instance. This report is produced automatically for each of the instances in turn; the user only has to answer "yes" or "no" for each question. The user may quit at any time by answering with "Q"; this has the affect of only extracting expressions that the user has answered "yes" to up to the point of quitting; all further expressions are left unchanged. If the user answers "yes" to any of the answers, HaRe also asks the user for the name of the abstraction and proceeds to transform the code to include the abstraction and the calls. This allows for cases where the user may not want to replace all instances of a particular clone, but only some of them; this behaviour is particularly useful if the clone detector reports clones classes of a large size.

6.4.3 Choosing the Location of the Abstracted Function

In Haskell, there are two cases to consider when placing the extracted function. The first is in a single-module project where the extracted function can be placed anywhere in the module. The second is a multiple-module project, where insertion of the extracted function in one of the constituent modules may introduce circular inclusions. Consider the code blocks in Figure 31; this figure

(A)

```
showTreeIndent :: Int -> Tree -> String
showTreeIndent m (Leaf c n)
  = spaces m ++ show c ++ " " ++ show n ++ "n"
showTreeIndent m (Node n t1 t2)
  = showTreeIndent (m+4) t1 ++
    spaces m ++ "[" ++ show n ++ "]" ++ "n" ++ showTreeIndent (m+4) t2
```

Figure 30: Two code fragments showing code instances for step-by-step removal

```
module A where
import B

emitJump p j i =
  emitByte p (j) >|>
  emitByte p (show l) >|>
  emitByte p (show h)
 where
   (h,l) = i ‘divMod‘ 256
```

```
module B where

emitOp12 p op i =
   case (-i) ‘divmod‘ 256 of
    (0,l) -> emitByte p (op ++ "_N1") >|>
             emitByte p (show l)
    (h,l) -> emitByte p (op ++ "_N2") >|>
             emitByte p (show h)

emitByte x y = ...
```

Figure 31: A multiple-module project demonstrating clone instances

```
module A where
import B

divMod256 i = i ‘divMod‘ 256
emitJump p j i =
  emitByte p (j) >|>
  emitByte p (show l) >|>
  emitByte p (show h)
 where
   (h,l) = divMod256 i
```

```
module B where
import A (divMod256)

emitOp12 p op i =
   case divMod256 (-i) of
     (0,l) -> emitByte p (op ++ "_N1") >|>
              emitByte p (show l)
     (h,l) -> emitByte p (op ++ "_N2") >|>
              emitByte p (show h)

emitByte x y = ...
```

Figure 32: A multiple-module project demonstrating circular inclusions

demonstrates a multiple-module project where we have, on the left hand side, a module that imports the module B, which is shown on the right hand side. The highlighted expressions in the figure show that the HaRe clone detector has discovered some cloned expressions. If the user has decided they would like to replace these sub-expressions with an abstraction the question is where does HaRe place the abstraction for these sub-expressions? In Figure 32, HaRe has placed the abstraction in the module that contains the instance of the first duplicated sub-expression that is selected for extraction. The problem here is that a cyclic dependancy has been introduced: module A imports module B and B imports module A. Cyclic inclusions must be avoided during the transformation due to the fact that transparent compilation of mutually-recursive modules is not supported by the current Haskell compilers/interpreters; although mutually-recursive modules are a part of the Haskell 98 standard. A possible solution to this problem is to put all the abstractions from a clone detection process into a separate module, but this then risks having problems when the abstractions depend on other functions themselves. The HaRe clone detector solves this problem by performing an analysis over the project under clone detection. If it is safe to place the abstraction into the module where the highlighted expression occurs, then this is performed. Otherwise, the abstraction is placed in the first module that does not introduce circular inclusions. In the case that expressions over multiple-modules are transformed into calls to the abstraction, the import relations of the modules containing those expressions are modified to import the module containing the abstraction. Suppose the module containing the abstraction is A and the module importing the abstracted module is B, the design issues for placing the abstraction are as follows:

- If module A has an explicit export list, the abstraction name is added to the export list.
- If module B has an explicit import list for module A, then the abstraction name is added to the explicit list of imports.

Otherwise an `import` statement is added to module B so that it imports module `A` with the abstraction name added to the list of imports for module A.

6.4.4 Calculating the Abstraction

The abstraction is calculated only when the transformation process can determine which sub-expressions can be passed in as an argument. This is done by comparing the identified expressions for transformation from the clone class. If, during the comparison, the sub-expressions are different, then they can be passed into the abstraction as a parameter. If the sub-expressions are the same (either as literals or identifiers referring to the same top-level identifier) then they do not need to passed in as arguments. Consider the following three clones:

```
(p,n+m) : alphaMerge xs ys
(p,n) : alphaMerge xs ((q,m):ys)
(q,m) : alphaMerge ((p,n):xs) ys
```

The transformation stage determines that from the three clones, the expressions `(:)` and `alphaMerge` occur in the same place in each expression. All other sub-expressions must be passed in as an argument. This process is called "anti-unification" as it is finding the least general generalisation of a set of expressions.

6.4.5 Abstracting Monadic Sequences

There are two cases to consider when abstracting duplicated monadic sequences. The first is where the abstraction contains pattern bindings that will be used in the remainder of the function containing the clone instance. The second is where the pattern bindings are used only in the clone. If any pattern bindings are needed by the function then they must be returned by the abstraction, and a new pattern binding is created to call the abstraction. Consider, for example (where the highlighted expressions show the clones):

```
f = do
   x <- return 565; y <- k 56
```

```
      putStrLn (y ++ show x)


   h = do
       a <- return 1111; b <- k 56
       putStrLn (b ++ show b)
       return 4
```

A new abstraction is created with a `return` statement, returning the values of the pattern bindings as a tuple in the abstraction:

```
   abs1 p_1
     = do
           x <- return p_1
           y <- k 56
           return (x,y)
```

The clone instances are then replaced with pattern bindings, so that the patterns bound in the abstraction can be further used in the program:

```
    f = do
         (x,y) <- abs1 565
         putStrLn (y ++ show x)


   h = do
          (a,b) <- abs1 1111
          putStrLn (b ++ show b)
          return 4
```

If the abstraction contains no pattern bindings, or the pattern bindings in the abstraction are not needed in the program, then no `return` statement is added to the abstraction. If only a subset of the variables are needed, then only these values are returned. Consider that HaRe has detected the following highlighted clones:

```
 f = do
       putStrLn (show 56 ++ " " ++ show 42)



h = do
      putStrLn (show 42 ++ " " ++ show 56)
      return 4
```

The following abstraction is created:

```
abs p_1 p_2 = putStrLn (show p_1 ++ " " ++ show p_2)
```

The clone instances are then replaced with calls to the abstraction:

```
f = do
      abs 56 42



h = do
      abs 42 56
      return 4
```

6.5 Summary

This chapter presented clone analysis and elimination as part of the HaRe infrastructure. In particular, two stages of clone removal were presented. The first stage as discussed in Section 6.3 was an automatic clone detection system, where expressions in a multiple-module project are compared against each other. Clones are reported to a text file. The second stage, as discussed in Section 6.4 was a transformation stage that allows the user to highlight a particular instance of a clone. HaRe then steps though the clones in the clone class, asking the user whether or not they would like to replace the clone with a call to an abstraction. The identified clone instances are then replaced with a call to an abstraction. The location of the abstraction is discussed in Section 6.4.3. Finally monadic clone elimination was discussed in Section 6.4.5.

Chapter 7

A Refactoring Case Study

The aim of this chapter is to present a case study of refactoring; the application of functional refactoring to a large-scale Haskell program is investigated and reported here.

The outline of the approach towards improving support for refactoring functional languages is also presented. The Haskell project that is chosen for refactoring is `nhc` [100]: a Haskell 98 compiler system written by members of the functional programming group at York University. `nhc` is significantly large to demonstrate the power of HaRe and it is also written using the Haskell 98 standard allowing HaRe to refactor the nhc source code. This chapter also makes use of the notion of a *refactorer* to denote the person performing the refactorings (as opposed to the refactoring software system).

The aims of the refactorings performed in the remainder of this chapter are as follows:

- Refactoring helps the refactorer to *understand* the program by transforming it incrementally into a program which conforms to their idiomatic style rather than the original programmer's.

- Refactoring is used to *simplify* the program and to make it more amenable to change: in this case extending the compiler in the future to deal with the Haskell Prime standard [21], say.

- Changing the structure of a program may also change its efficiency. The compiler is usually orthogonal to this, as using a compiler like GHC may perform optimizations on the code by applying similar transformations at compile time. A different Haskell compiler may not do these optimisations, however, and therefore refactoring may change the efficiency of the program. It is also interesting to note that having said this, refactoring may well improve a program's efficiency, but refactoring may also just as easily reduce it.

The chapter starts with a case study of refactoring `nhc`. In Section 7.2 we present some refactorings that are the subject of future work. Finally, in Section 7.3 we extend our refactoring case study by applying the refactorings defined in this thesis to a simple expression processing example.

7.1 Refactoring Sequence

The example was refactored in a series of steps, which are presented here in exactly the order in which they were performed. Upon reflection the refactorings might be performed differently, or in a different order. Discussion of future work follows in Section 7.2. The following refactorings are used in this chapter:

- Introduce a new definition, as defined by Li [67].
- Generalize a definition, as defined by Li [67].
- Folding, as defined in Section 4.1

Most of the rationale for the refactorings in this chapter was in terms of code understanding: putting units that would fit better together as a single named entity, for instance, or choosing names that more accurately reflected the action of a piece of code. The refactoring examples in stage one concentrate on a few specific cases for naming expressions that are of a large size. Reducing and abstracting them, therefore, was performed in order to increase the program's readability and understanding.

7.1.1 Stage 1

1. At first glance over some of the *core* files of nhc, a number of obvious applications for immediate refactoring became apparent. There are a number of expressions that can be simplified by introducing a new definition. The first example is within the function `addInstMethod` in *IntState.hs*:

```
addInstMethod :: TokenId -> TokenId -> TokenId -> NewType -> Id
              -> a -> IntState -> (Int,IntState)


addInstMethod  tidcls tidtyp tidMethod nt iMethod down
              state@(IntState unique rps st errors) =
  case lookupIS state iMethod of
   Just (InfoMethod u tid ie fix nt' (Just arity) iClass) ->
    (unique,IntState (unique+1) rps (addAT st (error "adding twice!")
     unique (InfoIMethod unique (mkQual3 tidcls tidtyp tidMethod) nt
     (Just arity) iMethod)) errors)
```

A further probing of the module shows that the application of `InfoIMethod` is used in many places. The rationale behind this and the next step, therefore, is to somehow lift the application of `InfoIMethod` to the top level, and abstract over certain entities within it and then perform a fold against it; the aim here is to remove some duplicated code and to make the code more amenable to change in the future and to allow the code to be understood easier.

A new definition, called `mkQualified`, is introduced for the expression `(mkQual3 tidcls tidtyp tidMethod)`. In addition to this, a new definition, called `infoMethod`, is introduced for the expression `InfoIMethod unique mkQualified nt (Just arity) iMethod`. This results in the following:

```
addInstMethod  tidcls tidtyp tidMethod nt iMethod down
              state@(IntState unique rps st errors) =
 case lookupIS state iMethod of
  Just (InfoMethod u tid ie fix nt' (Just arity) iClass) ->
   (unique,IntState (unique+1) rps (addAT st (error "adding twice!")
    unique (infoMethod)) errors)
     where
      mkQualified =  (mkQual3 tidcls tidtyp tidMethod)
      infoMethod  = InfoIMethod unique mkQualified
                     nt (Just arity) iMethod
```

2. As mentioned in the previous step, it seems that there are some places where it would better to share `InfoIMethod`, i.e. to lift it to the top level and then fold against it. However, in order to do this, the expression `Just arity` must be extracted to a definition within the `where` clause of `addInstMethod`. The reason for this is because in other instances of this expression throughout the program use expressions other than `(Just Arity)` as an argument to `InfoIMethod`. Therefore the expression must be generalised to take this into account. First a new definition is introduced called `justAritiy`:

```
infoMethod  = InfoIMethod unique mkQualified
                nt justArity iMethod
  where justArity = Just arity
```

and it is lifted by one level, so that `infoMethod` can also be lifted to the top level of the module. This is shown as follows:

```
addInstMethod  tidcls tidtyp tidMethod nt iMethod down
              state@(IntState unique rps st errors) =
 case lookupIS state iMethod of
  Just (InfoMethod u tid ie fix nt' (Just arity) iClass) ->
   (unique,IntState (unique+1) rps (addAT st (error "adding twice!")
    unique (infoMethod unique mkQualified nt justArity iMethod)) errors)
     where
       mkQualified =  (mkQual3 tidcls tidtyp tidMethod)
```

```
        justArity = Just arity


infoMethod unique mkQualified nt justArity iMethod
      = InfoIMethod unique mkQualified nt justArity iMethod
```

However, this has the drawback of creating the inverse of the previous abstraction step: lifting requires that the local definitions are to be passed in as formal arguments, which makes the call to `infoMethod` hold the same complexity as it did before step 2.

3. Folding is then performed over `infoMethod` converting an instance of the expression to a call to `infoMethod` within the function `updInstMethodNT`:

```
...
localAddAt
                 =   addAT st fstOf i
                         (infoMethod u tid nt annots iMethod)
...
```

Thus eliminating some duplicate code and introducing a useful abstraction. Observing this folding, it becomes apparent that an expression referring to the constructor `IntState` occurs frequently throughout the module. Therefore a new definition is introduced for `(IntState unique rps localAddAt errors)` occurring within `updInstMethodNT` and it is immediately lifted to the top-level. A fold is then performed over `intState`, transforming 12 instances of the application of `IntState` into calls to `intState`.

4. Continuing with the rationale of making large complex functions more understandable by abstracting them, the refactoring journey continues with looking at the module *Derive.hs*. Particularly the function `oneStep`:

```
oneStep state given
        ((cls_con@(cls,con),(free,ctxs)),pos_types@(pos,types)) =
 let (NewType _ _ data_ctxs _) = ntI (dropJust (lookupIS state con))
 in
  case ( sort
       . ctxsReduce state
       . (map (mapSnd mkNTvar) data_ctxs ++)
       . concatMap (ctxsSimplify [] state given)
       . map (\ nt -> TypeDict cls nt [(0,pos)])) types of
    ctxs -> ((cls_con,(free,map (mapSnd stripNT) ctxs)),pos_types)
```

Within `oneStep` a new definition is introduced, called `sortSimplifyCtxs`, for the expression:

```
(concatMap (ctxsSimplify empty state given))
```

A new definition is introduced within *Derive.hs*, called `toTypeDict` in the function `oneStep` for the expression:

```
(map (\ nt -> TypeDict cls nt [(0, pos)]))
```

A new definition is introduced within *Derive.hs*, called `toMkNTVar` in the function `oneStep` for the left section:

```
(map (mapSnd mkNTvar) data_ctxs ++)
```

This produces the following definition for `oneStep`:

```
oneStep state given
        ((cls_con@(cls,con),(free,ctxs)),pos_types@(pos,types)) =
 let (NewType _ _ data_ctxs _) = ntI (dropJust (lookupIS state con))
     sortSimplifyCtxs
         =   sort .
                 ((ctxsReduce state) .
                      ((toMkNTVar) .
                           (simplifyCtxs .
                                toTypeDict)))
```

```
           where
             toTypeDict
                 = (map (\ nt -> TypeDict cls nt [(0, pos)]))
             simplifyCtxs
                 = (concatMap (ctxsSimplify empty state given))
             toMkNTVar = (map (mapSnd mkNTvar) data_ctxs ++)
```

5. A new definition is introduced within *Derive.hs*, called `fstAndLstConstrs` in the function `startDeriving` for the expression:

   ```
   fstAndlst . constrsI
   ```

   The definition `types` within the first equation of `startDeriving` is lifted to the top-level, as it also appears as another definition in the second equation. The definition in the second equation is renamed to `localTypes` and folded against the lifted definition of `types`. The local definition of `localTypes` is then eliminated because it is no longer used.

6. In the module *Case.hs* there is an opportunity to introduce an as-pattern called `funs` for the pattern `(Fun args a (i:is))` within the function `matchFun`. This converts the expression `(Fun args a (i:is))` into `funs` as follows:

   ```
   matchFun :: Pos -> a -> [Fun Id] -> CaseFun PosLambda
   matchFun pos fun (funs@(Fun args a ((i : is))))
      = (caseUniques args) >>>=
           (\ iargs ->
             (caseNoMatch
                 ("Pattern match failure in function at " ++
                     (strPos pos))
                   pos) >>>=
                  (\ nomatch ->
                     let vars = map (\ (a, i) -> (getPos a, i)) iargs
   ```

(A)

```
startDeriving tidFun state (con,(pos,cls))
 | checkClass state cls tBounded =
  case lookupIS state con of
    Just conInfo ->
      let (NewType free empty ctx _) = ntI conInfo
          fstAndlst [] = []
          fstAndlst [x] = [x]
          fstAndlst xs = [head xs,last xs]
          fstAndLstConstrs = fstAndlst . constrsI
      in (((cls,con),(free,map (pair cls) free ++ ctx)),
         (pos,(types (.) snub concatMap init NewType ntI
          state fstAndLstConstrs conInfo)))
```

(B)

```
startDeriving tidFun state (con,(pos,cls)) =
  case lookupIS state con of
    Just conInfo ->
      let (NewType free [] ctx _) = ntI conInfo
          localTypes = types (.) snub concatMap init NewType
                       ntI state constrs conInfo
          constrs = constrsI
      in (((cls,con),(free,map (pair cls) free ++ ctx)),(pos,localTypes))
```

Figure 33: A code fragment within *Derive.hs* that has potential for refactoring

```
                        in (match (map (uncurry PosVar) vars) funs
                             nomatch) >>>=
                            (\ exp ->
                             unitS (PosLambda pos empty vars exp))))
```

7.1.2 Stage 2

This stage is concerned with the example in Figure 33 taken from *Derive.hs*; it is meant to be a separate stage to that of stage 1, although both stages could have been performed independently of each other. At first glance at the code in the figure, it seems that the two equations of `startDeriving` (defined in a local scope within the program) are very similar (the two equations are separated in the figure and labeled as *(A)* and *(B)* respectively). There are little differences between

(A)

```
startDeriving tidFun state (con,(pos,cls))
 =
  case lookupIS state con of
    Just conInfo ->
      let (NewType free empty ctx _) = ntI conInfo
          fstAndlst [] = []
          fstAndlst [x] = [x]
          fstAndlst xs = [head xs,last xs]
          fstAndLstConstrs = fstAndlst . constrsI
      in (((cls,con),(free,map (pair cls) free ++ ctx)),
         (pos,(types (.) snub concatMap init NewType ntI
          state constrs conInfo)))
```

(B)

```
startDeriving tidFun state (con,(pos,cls)) =
  case lookupIS state con of
    Just conInfo ->
      let (NewType free [] ctx _) = ntI conInfo
          localTypes = types (.) snub concatMap init NewType
                       ntI state constrs conInfo
          constrs = constrsI
      in (((cls,con),(free,map (pair cls) free ++ ctx)),(pos,localTypes))
```

Figure 34: The similarities between the two equations of `startDeriving`

(A) and *(B)* in that if we remove the guard from *(A)* and rename the expression `fstAndlst . constrsI` to `constrsI` (and therefore eliminating `fstAndlst` within the `where` clause of *(A)*) the two equations now become identical, as shown in Figure 34.

However, obviously performing the transformations as described above does not preserve the behaviour of `startDeriving`: the guard predicate is crucial in determining whether to evaluate `constrsI` or `fstAndLst . constrsI` within the definition of `types` in both equations. What follows is a series of refactorings that remove the second equation of `startDeriving` altogether. Some of the refactorings are already implemented refactorings within HaRe and some are refactorings that perhaps ought to have been implemented for HaRe and, are instead, performed manually (with an indication in the text).

1. At first glance over the code in block *(A)* of Figure 33, the definitions for `fstAndlst` and `fstAndLstConstrs` would be better positioned in a `where` clause for code clarity. Therefore the two `let` definitions are converted to a `where` clause and lifted by one level so that they scope the right-hand-side of the first equation of `startDeriving`.

2. A definition is manually introduced in the `where` clause of the first equation as follows:

   ```
   pred :: Bool -> a
   pred x = error "Function \"pred\" called - "
                  ++ "this is not yet defined"
   ```

   `pred` will later be used to model the behaviour of the guard predicate `checkClass state cls tBounded`. There is currently no transformation in HaRe that allows one to introduce a completely new definition like this. Perhaps it would have been worthwhile to implement a refactoring that inserts a definition into a selected scope. The refactoring could take a name as a parameter and the definition's type, introducing a skeleton function, as in the example of `pred` above. The advantage of this is that often programmers write code incrementally, first inserting the skeleton of a function say, and then adding pattern matching in a later stage of development, and therefore supporting top-down development.

3. New pattern matches are then introduced for `pred` using the introduce patterns refactoring (as described in Section 5.6). This transforms `pred` into the following:

   ```
   pred :: Bool -> a
   pred x@False = error "Function \"pred\" called - "
                   ++ "this is not yet defined"
   pred x@True = error "Function \"pred\" called - "
                   ++ "this is not yet defined"
   ```

   Setting `pred` up for further refactoring for modeling the guard behaviour.

4. A new parameter called `constrsI` is added to the definition of `pred`, producing the following:

   ```
   pred :: b -> Bool -> a
   pred constrsI x@False = error "Function \"pred\" called - "
                   ++ "this is not yet defined"
   pred constrsI x@True = error "Function \"pred\" called - "
                   ++ "this is not yet defined"
   ```

5. The right-hand-side of the first equation in `pred` is manually replaced with a call to `constrs constrsI`, as follows:

   ```
   pred :: b -> Bool -> a
   pred constrsI x@False = constrs constrsI
   pred constrsI x@True = error "Function \"pred\" called - "
                   ++ "this is not yet defined"
   ```

This is currently not a refactoring in HaRe. However, it may be a useful transformation to add, replacing one expression with another. The only problem was that the refactoring could not guarantee that the two expressions had the same behaviour, but perhaps this would not be the user's intent (as in this example). It is important to note that this transformation is not a refactoring as it does not preserve the program's behaviour. This process is known as refinement and not behaviour preservation, as it introduces behaviour that led to an error previously.

6. The right-hand-side of the second equation in `pred` is manually replaced with a call to `fstAndLstConstrs constrsI`, as follows:

```
pred :: b -> Bool -> a
pred constrsI x@False = constrs constrsI
pred constrsI x@True = fstAndLstConstrs constrsI
```

   Again, there is no refactoring in HaRe to do this transformation.

7. The next step is to make use of the newly introduced definition, `pred` in the first equation of `startDeriving`.

```
fstAndLstConstrs constrsI
```

   Is manually replaced with the following expression:

```
pred constrsI (checkClass state cls tBounded)
```

   This is another example of replacing one expression with another. Here we pass the guard predicate in as the second parameter to `pred` therefore allowing to determine the value passed into `types`.

8. The guard is then manually eliminated from the first equation of `startDeriving` and the second equation is commented out. Figure 35 shows the refactored definition of `startDeriving`.

```
startDeriving tidFun state (con,(pos,cls)) =
  case lookupIS state con of
    Just conInfo ->
      let (NewType free empty ctx _) = ntI conInfo

      in (((cls,con),(free,map (pair cls) free ++ ctx)),
          (pos,(types (.) snub concatMap init NewType ntI state
          (pred constrsI (checkClass state cls tBounded))
           conInfo)))
    where
      pred :: b -> Bool -> a
      pred constrsI x@False = constrs constrsI
      pred constrsI x@True = fstAndLstConstrs constrsI
      pred constrsI x = error x

      fstAndlst head last [] = []
      fstAndlst head last [x] = [x]
      fstAndlst head last xs = [head xs,last xs]
      constrs constrsI = constrsI
      fstAndLstConstrs = (fstAndlst head last) . constrsI
```

Figure 35: The refactored definition of `startDeriving`

7.1.3 Stage 3 - Clone Removal

The third, and final, stage of the case study was to run a clone analysis over the `nhc` project. For the case study, the clone analyzer was configured so that there was no limit on the size of clone classes and the minimum number of tokens appearing in an expression is 5. Due to space restrictions in this thesis the clone analysis was only run over the `Main` module to show the clone analysis as a proof of concept. The results of the clone analysis can be found in Appendix E.

The first step was to extract some of the clones reported into abstractions. Appendix F shows an extract of some of the clone classes that were considered for extraction. The three clone classes are separated by calls to different top-level identifiers. The first class makes calls to the identifier `getRenameTableIS`; the second calls `getSymbolTableIS` and the third calls `getSymbolTable`. The clone analysis does not identify two or more code fragments to be identical if they refer to different top-level identifiers. Therefore the code in these cases must be extracted into separate abstractions and then folded so that they can all become instances of the same function.

First, the expressions identified in the last clone class are extracted first as this class contains the most clone occurrences. The expression `mixLine (map show (listAT (getSymbolTable state)))` is selected within HaRe and *extract expression* is selected from the HaRe drop-down menu. After some small analysis, HaRe then prompts for each expression in the clone class, whether the expression should be extracted or not. It seems appropriate for all expressions to be extracted. HaRe then creates an abstraction (here, `state` is a local variable that must be passed in as a parameter):

```
abs_1 p_1
    = mixLine (map show (listAT (getSymbolTable p_1)))
```

And replaces the expressions in the clone class with:

```
abs_1 state
```

The name is chosen by the refactoring automatically; however the choice of the uninformative name can easily be fixed by a renaming. The same is performed for the clone expressions identified in the two former clone classes of Appendix F. The expressions occuring in the two remaining classes are selected in turn, and extracted into calls to the following abstractions. For example, the expressions of the form:

```
mixLine (map show (listAT (getRenameTableIS importState)))
```

are converted into calls as in the following

```
abs_2 importState
```

Likewise, expressions of the form:

```
mixLine (map show (listAT (getSymbolTableIS importState)))
```

are converted into calls as in the following

```
abs_3 importState
```

The following abstractions are introduced at the top of the module:

```
abs_2 p_1
  = mixLine (map show (listAT (getSymbolTableIS p_1)))


abs_3 p_1
  = mixLine (map show (listAT (getRenameTableIS p_1)))
```

It now makes sense to convert `abs_3` into a call to `abs_2` in the following way:

- The definition of `abs_2` is generalised so that `getSymbolTableIS` is made a parameter:

  ```
  abs_2 p_0 p_1
   = mixLine (map show (listAT (p_0 p_1)))
  ```

(this also update all calls to `abs_2 localState` with `abs_2 getSymbolTableIS localState`).

- A fold is then performed against the definition of

  ```
  abs_2 p_0 p_1
       = mixLine (map show (listAT (p_0 p_1)))
  ```

  so that `abs_3` now calls `abs_2`:

  ```
   abs_3 p_1
       = abs_2 getRenameTableIS p_1
  ```

A further extraction of other clones reported by the analysis will remove even more duplicate entities, but for the purposes of illustration only a few cases have been summarised in this section. The last clone class of Appendix E is particularly interesting in that, although the clone class has 26 occurrences, none of them appear to be true duplicates, or duplicates that are worth extracting. The main problem is that the analyzer can not distinguish between useful and non-useful clone entities. Indeed, the analysis also compares expressions of the form

```
f e1 e2
```

with

```
f e1 e2 e3
```

The reason for this is mainly due to the fact that the clone analysis is tree-based. `f e1 e2` is actually represented as `(f e1) e2` and, if `e2` is a variable that is not declared at the top-level, it will consequently match against anything.

Monadic clone detection could not be performed in this case study. `nhc` only has a small amount of monadic code and that code is not suitable for clone extraction. Looking through *MainNew.hs*, the following is an example of the monadic code found there:

```
  (state  -- :: IntState
     ,fixState    -- :: FixState
     ,gcode)      -- :: [Gcode]
            <- return (gcodeFix flags state fixState gcode)
    pF (sGcodeFix flags) "G Code (fixed)" (concatMap (strGcode state) gcode)

    (gcode        -- :: [Gcode]
     ,state)      -- :: IntState
            <- return (gcodeOpt1 state gcode)
    pF (sGcodeOpt1 flags) "G Code (opt1)" (concatMap (strGcode state) gcode)

    (gcode        -- :: [Gcode]
     ,state)      -- :: IntState
            <- return (gcodeMem (sProfile flags) state gcode)
    pF (sGcodeMem flags) "G Code (mem)" (concatMap (strGcode state) gcode)
```

In the above, `sGcodeFix`, `sGcodeOpt1` and `sGcodeMem` are all different top level identifiers and are therefore not identified as code clones.

7.2 Future Refactorings

This section discusses the various observations that were made to the nhc source code in respect to designing new refactorings for future work.

7.2.1 Name a `type`

A problem that kept occurring in nhc was the use of obfuscated type signatures, for example Figure 36 (taken from Derive.hs) introduces two complex types that leave the refactorer puzzled as to what is going on. This problem continues with the recurring pattern of types as shown in the code blocks in Figure 38.

It seems obvious from this that it would be beneficial in future work to add a new refactoring to HaRe to allow the identification of an instance of a type, and have HaRe either refactor all instances of that type into a type synonym, or only identify some instances or one instance where the type folding would be useful.

The following gives a brief summary as to how this refactoring would be invoked by the user, and any particular conditions that arises:

- The user selects a particular type occurring within a type signature and inputs a new name for the type into HaRe. Alternatively, the user could select the right-hand-side of an already defined type synonym, and HaRe could step through each type that matches the selected type in question.
- As the analogue to folding for *expressions*, this refactoring is folding for *types*. Any polymorphic types occurring in the type must be passed in as a type parameter.
- `type` synonyms cannot be used in the subject of `instance` declaration, therefore this refactoring will not work in `instance` declarations.

Figure 37 shows how the example code in Figure 36 could be simplified with a type synonym. Looking through the code and making my own judgement, it seems the type `[(Id,[(Pos,Id)])]` refers to the class that derives a list of type constructors and their source code position. Introducing the type synonym `type Derived = [(Id,[(Pos,Id)])]` and folding all occurrences of `[(Id,[(Pos,Id)])]` against it results in much easier code to understand as the type in question refers to types instances that have to be derived.

7.2.2 Transforming Between Guards and `if..then..else`

This refactoring could be used to great effect in eliminating complex nested `if..then..else` constructs that appear as the result of badly designed functions. Figure 39 shows an example of where the programmer has introduced a nested `if..then..else` and as a result, the code in Figure 39 is very difficult to understand. Figure 40 shows how converting the nested `if..then..else` into guards not only simplifies the function but also allows further refactorings to be performed easily. Converting the code to guards in this manner makes the underlying predicates of the nested `if` expressions much clearer. This refactoring has

(A)

```
derive :: ((TokenId,IdKind) -> Id) ->
          IntState                 ->
          [(Id,[(Pos,Id)])]        ->
          Decls Id                 ->
          Either [String] (IntState,(Decls Id))
derive = ...
```

(B)

```
doPreWork :: [(Id,[(Pos,Id)])] -> [(Id,(Pos,Id))]
doPreWork d = ...
```

Figure 36: Two code fragments showing instances for name a `type`

(A)

```
type Derived = [(Id,[(Pos,Id)])]
derive :: ((TokenId,IdKind) -> Id) ->
          IntState                 ->
          Derived       ->
          Decls Id                 ->
          Either [String] (IntState,(Decls Id))
derive = ...
```

(B)

```
doPreWork :: Derived -> [(Id,(Pos,Id))]
doPreWork d = ...
```

Figure 37: Two code fragments showing the refactored code for name a `type`

(A)

```
work :: [(Id,(Pos,Id))] -> [(((Int,Int),([Int],[(Int,Int)])),(Pos,[NT]))]
work preWork = ...
```

(B)

```
deriveOne :: IntState
          -> ((TokenId,IdKind) -> Int)
          -> (((Int,Int),([Int],[(Int,Int)])),(Pos,a))
          -> b
          -> IntState
          -> (Decl Int,IntState)
deriveOne = ...
```

(C)

```
oneStep ::  IntState
        -> [ ((Int,Int),([Int],[(Int,Int)])) ]
        -> ( ((Int,Int),([Int],[(Int,Int)])) , (Pos,[NT]) )
        -> ( ((Int,Int),([Int],[(Int,Int)])) , (Pos,[NT]) )
oneStep = ...
```

Figure 38: Further code fragments showing instances for name a `type`

(A)

```
iLex :: LexState -> Int -> [PosTokenPre] -> [PosToken]
iLex s i [] = []
iLex s i ((f,p,c,t):pt) =
  seq p $
  if c > i then
    piLex f s i p t pt
  else if c == i && i /= 0 && t /= L_in then
    seq p' $ (p',L_SEMI',s,pt) : piLex f s i p t pt
  else if c == 0 && i == 0 then
    piLex f s i p t pt
  else
    seq p' $ (p',L_RCURL',s,pt) : iLex s' i' ((f,p,c,t):pt)
  where
  (_:s'@(i':_)) = s
  p' = insertPos p
```

Figure 39: A code fragment showing an instance for `if..then..else` to guards

already been implicitly implemented as a part of folding (described in Section 4.1) and would really benefit from being a separate refactoring in its own right. There is no restriction with converting `if..then..else` expressions to guards, however the converse requires there to be an `otherwise` case in the guard.

7.2.3 Nested `Case` Expression Elimination

Figure 41 shows an example of a nested `case` expression that could be difficult for a maintainer of `nhc` to understand. Although, through first observations, there may be only a few cases where this refactoring is useful. Converting nested `case` expressions into a single top-level `case` expression would be very beneficial to help reduce the propagation of bugs in a pattern match, and to also help to understand the logic of a particular function. Suppose we have the following `case` expression actually taken from a parser implementation written in Haskell (not taken from `nhc`):

```
case s of
```

(A)

```
iLex :: LexState -> Int -> [PosTokenPre] -> [PosToken]
iLex s i [] = []
iLex s i ((f,p,c,t):pt)
  | c > i = seq p $ piLex f s i p t pt
  | c == i && i /= 0 && t /= L_in = seq p $ seq p' $ (p',L_SEMI',s,pt)
                                     : piLex f s i p t pt
  | c == 0 && i == 0 = seq p $ piLex f s i p t pt
  | otherwise = seq p $ seq p' $ (p',L_RCURL',s,pt)
                  : iLex s' i' ((f,p,c,t):pt)
  where
  (_:s'@(i':_)) = s
  p' = insertPos p
```

Figure 40: A code fragment showing the conversion of `if..then..else` to guards

```
(TyVar x) -> case t of
               (TyVar y) -> parseVar y
               otherwise
                 -> error "Parse error: Identifier expected!"
(x :->: y) -> case t of
             (TyVar y) -> parseVar y
             (x' :->: y') -> parseVar x' `seq` parseVar y'
             otherwise
                -> error "Parse error: Identifier expected!"
```

The refactoring would unify the multi-layer `case` expressions into a single one, as follows:

```
case (s,t) of
  (_,TyVar y) -> parseVar y
  (x :-> y,x' :->: y') -> parseVar x' `seq` parseVar y'
  otherwise -> error
```

This has the advantage of introducing sharing: the two `case` expressions that produced an `error` are collapsed into one. The two `case` expressions where

(A)

```
updInstMethodNT :: TokenId -> TokenId -> Int -> NewType -> Int
                -> a -> IntState -> IntState

updInstMethodNT tidcls tidtyp i nt iMethod  down
      state@(IntState unique rps st errors) =
  case lookupAT st iMethod of
    Just (InfoMethod _ _ _ _ _ annots _) ->
      case lookupAT st i of
        Just (InfoIMethod u tid' _ _ _) ->
            let tid = mkQual3 tidcls tidtyp (dropM tid')
            in IntState unique rps
                (addAT st fstOf i (InfoIMethod u tid nt annots iMethod))
                errors
```

Figure 41: A code fragment showing an instance of a nested `case` expression

`doSomething` is evaluated also gets collapsed into one. Figure 42 shows how the nested `case` expressions can be collapsed into one single `case` expression. The above example, however, is not meaning preserving. The first example will fail for `undefined` for `s`, whereas `(undefined, TyVar y)` will succeed in the second example.

7.3 Refactoring an Expression Processing Example

In this section, we present a simple example for illustrating how the majority of the refactorings described in this thesis could be used. For the example, we design a very simple language; we then write a parser, evaluator and pretty printer for that language. As the application is being implemented, there are cases where the use of a refactoring tool greatly increases the productivity of the programmer, and improves the design of the program, making the succeeding implementation steps easier to perform. The following refactorings from this thesis are used in

(A)

```
updInstMethodNT :: TokenId -> TokenId -> Int -> NewType -> Int
                -> a -> IntState -> IntState

updInstMethodNT tidcls tidtyp i nt iMethod  down
      state@(IntState unique rps st errors) =
  case (lookupAT st iMethod, lookupAT st i) of
    (Just (InfoMethod _ _ _ _ _ annots _), Just (InfoIMethod u tid' _ _ _))
      ->
        let tid = mkQual3 tidcls tidtyp (dropM tid')
            in IntState unique rps
                (addAT st fstOf i (InfoIMethod u tid nt annots iMethod))
                errors
```

Figure 42: A code fragment showing an instance of a nested `case` expression eliminated into a `case` expression with a tuple pattern match

this example:

- Splitting (described in Section 3.3.1 on Page 69).
- Merging (described in Section 3.3.2 on Page 72).
- Folding (described in Section 4.1 on Page 78).
- Add a constructor (described in Section 5.2 on Page 111).
- Remove a constructor (described in Section 5.2 on Page 111).
- Add a field to a data type (described in Section 5.4 on Page 118).
- Introduce pattern matching (described in Section 5.6 on Page 122).

In addition to these, we also make use of the following refactorings from Li's [67] thesis:

- Renaming.
- Generalise a definition.

```
 1   module Parser where
 2   import Data.Char
 3
 4   data Expr = Literal Int | Plus Expr Expr
 5                deriving Show
 6
 7
 8   parseExpr :: String -> (Expr, String)
 9   parseExpr (' ':xs) = parseExpr xs
10   parseExpr ('+':xs) = (Plus parse1 parse2, rest2)
11                           where
12                             (parse1, rest1) = parseExpr xs
13                             (parse2, rest2) = parseExpr rest1
14   parseExpr (x:xs)
15      | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
16               where
17                 lit = parseInt xs
18                 parseInt :: String -> String
19                 parseInt [] = []
20                 parseInt (x:xs) | isNumber x = x : parseInt xs
21                                 | otherwise  = []
22   parseExpr xs = error "Parse Error!"
23
24   eval :: Expr -> Int
25   eval (Literal x) = x
26   eval (Plus x y) = (eval x) + (eval y)
```

Figure 43: A parser for a simple language

- Introduce a new definition.
- Add a new parameter.

The example starts with the very basics of implementing a language, parser and evaluator. The code for this is shown in Figure 43 on Page 177. The grammar for the language is described in a data type on Line 4; so far, the language only has the capacity to handle simple literals and applications of `Plus`. The function `parseExpr` is the parser for the language: taking a `String` and converting it into a tuple: the first element being the Abstract Syntax Tree for `Expr`, and the second the unconsumed input. `Expr` is an initial segment of the input, whatever is unconsumed and returned. To show this in practice, the following shows how the parser and evaluator can be invoked from the GHCi command line:

```
Prelude Parser> parseExpr "+ 1 2"
(Plus (Literal 1) (Literal 2),"")

Prelude Parser> eval (fst $ parseExpr "+ 1 2")
3
```

For reasons of simplicity, the language does not include parentheses (although this could easily be integrated into future versions) and `+` is not applied as an infix function, also the expressions only take positive integers. For the purpose of this example the expressions are given in a prefix format.

7.3.1 Step 1: Initial Implementation

With the basics of the parser and evaluator set up, the first step is to start integrating other constructs into the language. Therefore, we add the constructor `Mul` to `Expr` in order to represent the application of `*` in our programs. We do this by using the refactoring *add a constructor* (described in Section 5.2 on Page 111). The refactoring asks us for the name of the constructor and any parameters. We enter `Mul Expr Expr` and the refactoring adds this to the end of `Expr` and also generates additional pattern matching clauses to `eval`:

```
data Expr = Literal Int | Plus Expr Expr | Mul Expr Expr
addedMul = error "Added Mul Expr Expr to Expr"
...
eval :: Expr -> Int
eval (Literal x) = x
eval (Plus x y) = (eval x) + (eval y)
eval (Mul p_1 p_2) = addedMul
```

The refactoring also inserts a call to the automatically created equation `addedMul` which is easily replaced with actual functionality in the succeeding steps.

7.3.2 Step 2: Introduce Binary Operators

In the future it is hoped that the language will be able to handle any number of mathematical binary operators. In order to handle this design decision, we implement a new data type `Bin_Op` to handle binary operators, and a new constructor to `Expr` to handle this abstraction. In order to do this implementation, we first *remove* the constructors `Plus` and `Mul` (using the *remove a constructor* refactoring, defined in Section 5.2 on Page 111). The refactoring then automatically removes both constructors and their pattern matching:

```
   data Expr = Literal Int
   ...
   parseExpr ('+':xs) = (error "Plus removed from Expr"
                          {-Plus parse1 parse2, rest2-})
                        where
                          (parse1, rest1) = parseExpr xs
                          (parse2, rest2) = parseExpr rest1
   ...
   eval :: Expr -> Int
   eval (Literal x) = x
{- eval (Plus x y) = (eval x) + (eval y) -}
{- eval (Mul p_1 p_2) = (eval x) * (eval y) -}
```

The `Bin_Op` data type is then created with the constructors, `Mul` and `Plus`. A new function, called `eval_op` is then introduced, with a skeleton implementation, as follows:

```
   eval_op :: (Num a) => Bin_Op -> (a -> a -> a)
   eval_op x = error "Undefined Operation"
```

We then proceed to define the implementation for `eval_op`: by choosing *introduce pattern matching* (described in Section 5.6 on Page 122) from HaRe and selecting the argument `x` within `eval_op`, the refactoring produces the following:

```
eval_op :: (Num a) => Bin_Op -> (a -> a -> a)
eval_op p_1@(Mul) = error "Undefined Operation"
eval_op p_1@(Plus) = error "Undefined Operation"
eval_op _ = error "Undefined Operation"
```

All that is left to do for this stage is to replace the right-hand-sides of `eval_op` with `(*)` and `(+)` respectively and remove the redundant variable `p_1` introduced by the refactoring. The completed implementation so far is given in Figure 44.

7.3.3 Stage 3: House Keeping

The next stage is to do some tidying of our newly introduced type, `Bin_Op`. In particular, we need to define a constructor within `Expr` and modify the evaluator to call `eval_op` for the `Bin_Op` case.

To start, we *add a constructor* to `Expr` where HaRe also automatically adds a new pattern clause to `eval`:

```
data Expr = Literal Int | Bin Bin_Op Expr Expr
addedBin = error "Added Bin Bin_Op Expr Expr to Expr"
...
eval :: Expr -> Int
eval (Literal x) = x
eval (Bin p_1 p_2 p_3) = addedBin
```

The next step is then to *rename* (using the *rename* refactoring in HaRe) the variables in the introduced pattern match to something more meaningful:

```
eval :: Expr -> Int
eval (Literal x) = x
eval (Bin op e1 e2) = eval_op op (eval e1) (eval e2)
```

The call to `error` on the right hand side of `parseExpr` for the `'+'` case is then replaced with `Bin Plus parse1 parse2`. Multiplication is then introduced in the parser, by copying the `'+'` case into a `'*'` case, and substituting `Plus` for `Mul` on

```
module Parser where
import Data.Char

data Expr = Literal Int
             deriving Show

data Bin_Op = Mul | Plus

addedMul = error "Added Mul Expr Expr to Expr"

parseExpr :: String -> (Expr, String)
parseExpr (' ':xs) = parseExpr xs
parseExpr ('+':xs) = (error "Plus removed from Expr"
                   {-Plus parse1 parse2, rest2-})
                       where
                         (parse1, rest1) = parseExpr xs
                         (parse2, rest2) = parseExpr rest1
parseExpr (x:xs)
  | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
            where
              lit = parseInt xs
              parseInt :: String -> String
              parseInt [] = []
              parseInt (x:xs) | isNumber x = x : parseInt xs
                              | otherwise  = []
parseExpr xs = error "Parse Error!"

eval :: Expr -> Int
eval (Literal x) = x
{- eval (Plus x y) = (eval x) + (eval y) -}
{- eval (Mul p_1 p_2) = (eval x) * (eval y) -}

eval_op :: (Num a) => Bin_Op -> (a -> a -> a)
eval_op p_1@(Mul) = (*)
eval_op p_1@(Plus) = (+)
eval_op _ = error "Undefined Operation"
```

Figure 44: The parser implementation with the generality of binary operators expressed

the right hand side. The complete implementation, so far, is shown in Figure 45 on Page 183.

7.3.4 Stage 4: Generalisation

The next step is to do some *generalisation* and *folding*. As can be seen from Figure 45, two equations of `parseExpr` contain some duplicated code (this is highlighted in the figure). We eliminate this duplicate code, by first *introducing a new definition* (using the *introduce new definition refactoring in HaRe*) by highlighting the code on lines 20 - 22 from Figure 45. We enter `parseBin` as the name for the new expression, and HaRe introduces the following code:

```
...
parseExpr ('+':xs) = parseBin xs
...
parseBin xs = (Bin Plus parse1 parse2, rest2)
                     where
                       (parse1, rest1) = parseExpr xs
                       (parse2, rest2) = parseExpr rest1
```

The code highlighted in italics show how the refactoring has replaced the right hand side of the equation `parseExpr` with a call to `parseBin`. Obviously, the function `parseBin` should now be generalised so that the constructors `Plus` and `Mul` can be passed in as formal arguments. This will also allow us to *fold* (using *folding* as described in Section 4.1 on Page 78) the equation `parseExpr` defined on Line 15 of Figure 45 against the new definition `parseBin`. The following code illustrates this:

```
parseExpr :: String -> (Expr, String)
parseExpr (' ':xs) = parseExpr xs
parseExpr ('*':xs) = parseBin Mul xs
parseExpr ('+':xs) = parseBin Plus xs
parseExpr (x:xs)
```

```
module Parser where
import Data.Char

data Expr = Literal Int | Bin Bin_Op Expr Expr
          deriving Show

data Bin_Op = Mul | Plus deriving Show

addedMul = error "Added Mul Expr Expr to Expr"
addedBin = error "Added Bin Bin_Op Expr Expr to Expr"


parseExpr :: String -> (Expr, String)
parseExpr (' ':xs) = parseExpr xs
parseExpr ('*':xs) = (Bin Mul parse1 parse2, rest2)
                     where
                       (parse1, rest1) = parseExpr xs
                       (parse2, rest2) = parseExpr rest1
parseExpr ('+':xs) = (Bin Plus parse1 parse2, rest2)
                     where
                       (parse1, rest1) = parseExpr xs
                       (parse2, rest2) = parseExpr rest1
parseExpr (x:xs)
  | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
            where
              lit = parseInt xs
              parseInt :: String -> String
              parseInt [] = []
              parseInt (x:xs) | isNumber x = x : parseInt xs
                              | otherwise  = []
parseExpr xs = error "Parse Error!"

eval :: Expr -> Int
eval (Literal x) = x
eval (Bin op e1 e2) = eval_op op (eval e1) (eval e2)

eval_op :: (Num a) => Bin_Op -> (a -> a -> a)
eval_op p_1@(Mul) = (*)
eval_op p_1@(Plus) = (+)
eval_op _ = error "Undefined Operation"
```

Figure 45: The parser implementation with plus and multiplication

```
  | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
           where
             lit = parseInt xs
             parseInt :: String -> String
             parseInt [] = []
             parseInt (x:xs) | isNumber x = x : parseInt xs
                             | otherwise  = []
parseExpr xs = error "Parse Error!"

parseBin p_1 xs = (Bin p_1 parse1 parse2, rest2)
                    where
                      (parse1, rest1) = parseExpr xs
                      (parse2, rest2) = parseExpr rest1
```

This refactoring has allowed to keep the implementation consistent: there is now a separate evaluator for binary operators as well as a separate parser for binary operators; this allows for the code to be easily maintained in future versions.

7.3.5 Stage 5: Introduce Variables

We now add variables to our language by defining the `let` expression. In order to do this, we need to add the `Let` construct to our language `Expr` and then extend the parser to deal with the new construct. Having variables in our language also means that we need to be able to store bindings within an environment, and then use a `lookup` on that environment when we evaluate the Abstract Syntax Tree.

To perform this extension to our language, first we perform an *add constructor* refactoring to the definition of `Expr`, adding `LetExp String Expr` as an argument to the refactoring. The refactoring then introduces new pattern matching for `eval`, as follows:

```
data Expr = Literal Int | Bin Bin_Op Expr Expr
          | LetExp String Expr
addedBin = error "Added LetExp String Expr to Expr"
```

```
...
eval :: Expr -> Int
eval (Literal x) = x
eval (Bin op e1 e2) = eval_op op (eval e1) (eval e2)
eval (LetExp p_1 p_2) = addedLetExp
```

The `lookup` function is now defined, but while implementing this, we soon realise that `LetExp` in fact needs a new field added to it to be able to represent the expression bound to the variable. We therefore choose *add a field* from HaRe (defined in Section 5.4 on Page 118) and enter `Expr` as a parameter. This then modifies the implementation of `LetExp` and also adds a new variable to the pattern matching in `eval`, the following illustrates this:

```
data Expr = Literal Int | Bin Bin_Op Expr Expr
          | LetExp String Expr Expr
addedBin = error "Added LetExp String Expr to Expr"
...
eval :: Expr -> Int
eval (Literal x) = x
eval (Bin op e1 e2) = eval_op op (eval e1) (eval e2)
eval (LetExp n e e_2) = addedLetExp
```

We then modify the parser to cope with `Let` expressions in the language, and add a new constructor `Var String` to `Expr` (again, using the *add a constructor* refactoring. The full implementation up to this point is illustrated in Appendix I.1 on Page 266.

7.3.6 Stage 6: Merging

The next stage is concerned with implementing a pretty printer for our language. We do this by defining a function `prettyPrint` over the type `Expr` with a type signature, and choose the *introduce pattern matching* from HaRe. This produces the following:

```
prettyPrint :: Expr -> String
prettyPrint x@(Literal x) = error "Unable to pretty print!"
prettyPrint x@(Bin op e1 e2) = error "Unable to pretty print!"
prettyPrint x@(LetExp n e e_2) = error "Unable to pretty print!"
prettyPrint x@(Var n) = error "Unable to pretty print!"
prettyPrint x = error "Unable to pretty print!"
```

The implementation for `prettyPrint` is completed, and the same proceedure is repeated for a function `prettyBinOp` (including *introduce pattern matching*) in order to represent the pretty printing of binary operators. This gives us the following definitions:

```
prettyPrint :: Expr -> String
prettyPrint x@(Literal y) = show y
prettyPrint x@(Bin op e1 e2) = prettyPrintBinOp op
            ++ " " ++ (prettyPrint e1) ++ " " ++ (prettyPrint e2)
prettyPrint x@(LetExp n e e_2) = "let " ++ n ++ " = "
            ++ (prettyPrint e) ++ " in " ++ (prettyPrint e_2)
prettyPrint x@(Var n) = n
prettyPrint x = error "Unable to pretty print!"

prettyPrintBinOp :: Bin_Op -> String
prettyPrintBinOp x@(Mul) = "*"
prettyPrintBinOp x@(Plus) = "+"
prettyPrintBinOp x = error "Unable to pretty print binary operator"
```

To show how the pretty printer and parser work in practice, the following shows an example from the GHCi prompt:

```
Prelude Parser> parseExpr "let x + 1 1 x"
(LetExp "x" (Bin Plus (Literal 1) (Literal 1)) (Var "x"),"")

Prelude Parser> prettyPrint (LetExp "x" (Bin Plus (Literal 1)
                (Literal 1)) (Var "x"))
"let x = + 1 1 in x"


Prelude Parser> eval [] (LetExp "x" (Bin Plus (Literal 1)
                (Literal 1)) (Var "x"))
2
```

As can be seen, both `eval` and `prettyPrint` take an `Expr` as an argument. It would be nice to merge the two functions together so that it may be possible to pretty print and evaluate an abstract syntax tree simultaneously. This may lead a function that parses an input, and pretty prints and evaluates the output, as follows:

```
Prelude Parser> parse "let x + 1 1 x"
"The value of let x = + 1 1 in x is 2"
```

In order to implement this feature, we first *merge* the definitions of `prettyPrint` and `eval` together (the *merge* refactoring is defined in Section 3.3.2 on Page 72). We also move the definitions of `eval_op` and `prettyPrintBinOp` to a `where` clause of the newly merged `eval` function. We then merge these two functions as well, producing the code in Figure 46. Appendix I.2 on Page 268 shows the current state of the parser implementation up to the end of this stage.

7.3.7 Stage 7: Splitting

In the final stage of writing the parser, we extract the first element of the `parseExpr` function by *splitting* its result (using *splitting* as defined in Section 3.3.1 on Page 69). The reason we do this splitting is because, at the moment, `parseExpr` returns a tuple: the first component being the Abstract Syntax Tree for the parsed input, and the second component being the unconsumed input. In the final state of the parse, however, we do not want to return the unconsumed input, and therefore extract the first element of the parser into a new definition. We then use this new definition to call the original parser, eliminating the unconsumed input.

```
eval :: Environment -> Expr -> (String, Int)
eval env (Literal x) = (show x, x)
eval env (Bin op e1 e2) = ((fst (eval_op op)) ++ " "
                        ++ (fst $ eval env e1) ++ " "
                        ++ (fst $ eval env e2),
                        (snd $ eval_op op) (snd $ eval env e1)
                        (snd $ eval env e2))
                 where
                    eval_op :: (Num a) => Bin_Op -> (String, (a -> a -> a))
                    eval_op p_1@(Mul) = ("*", (*))
                    eval_op p_1@(Plus) = ("+",(+))
                    eval_op _ = error "Undefined Operation"
eval env (LetExp n e e_2) = ("let " ++ n ++ " = " ++ (fst $ eval env e)
                          ++ " in " ++ (fst $ eval env e_2),
                          snd $ eval (addEnv n e env) e_2)
eval env (Var n) = (n, snd $ eval env (lookUp n env))
```

Figure 46: The parser implementation with the generality of binary operators expressed

In order to do the splitting, we move the current definition of `parseExpr` into a `where` clause of a newly defined function called `parse`. We then extract the first component of the result of `parseExpr` using *splitting*, as follows:

```
parse :: String -> Expr
parse input = parseFst input
   where
    parseFst :: String -> Expr
    parseFst (' ':xs) = parseFst xs
    parseFst ('*':xs) = fst $ parseBin Mul xs
    parseFst ('+':xs) = fst $ parseBin Plus xs
    parseFst (x:xs)
     | isNumber x = Literal (read (x:lit)::Int)
               where
                 lit = parseInt xs


    parseFst (x:xs)
```

```
 | isChar x = case var of
             "let" -> LetExp (isName name) expr1 expr2
             x  ->  Var var
            where
              name = parseVar remainder
              (expr1, rest2) = parseExpr (drop (length name) remainder)
              expr2  = parseFst rest2

              remainder = (drop (length var) xs)
              var = x: (parseVar xs)



parseExpr :: String -> (Expr, String)
parseExpr (' ':xs) = parseExpr xs
parseExpr ('*':xs) = parseBin Mul xs
parseExpr ('+':xs) = parseBin Plus xs
parseExpr (x:xs)
 | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
            where
              lit = parseInt xs

parseExpr (x:xs)
 | isChar x = case var of
             "let" -> (LetExp (isName name) expr1 expr2, rest3)
             x  ->  (Var var, remainder)
            where
              name = parseVar remainder
              (expr1, rest2) = parseExpr (drop (length name) remainder)
              (expr2, rest3) = parseExpr rest2
              remainder = (drop (length var) xs)
              var = x: (parseVar xs)
```

```
parseExpr xs = error "Parse Error!"

parseBin p_1 xs = (Bin p_1 parse1 parse2, rest2)
                    where
                      (parse1, rest1) = parseExpr xs
                      (parse2, rest2) = parseExpr rest1
```

The same procedure is then repeated for `parseBin`, extracting out the first component of its result, and the expressions `fst $ parseBin Mul xs` within `parseFst` are transformed into calls to the extracted function `parseBinFst`. It is important to note that this results in a more efficient implementation than in the original program as evaluating and pretty printing now share the same variables. The result of the final parser implementation is shown in Appendix I.3 on Page 269.

7.4 Summary

This chapter presented a summary of a case study of refactoring for `nhc`. Stage 1 (described in Section 7.1.1) showed a set of rudimentary refactorings that were performed over a number of core files of `nhc`. In particular, the folding refactoring, as described in Section 4.1, was used after introducing definitions and generalising them. Stage 2 (described in Section 7.1.2) showed an example where refactoring can be used to great effect in showing a special case of clone elimination. A function where two of its (almost identical) equations was presented; refactoring was then performed to completely eliminate the need for the second equation. Stage 3 (described in Section 7.1.3) gave a summary of some clone analysis and elimination. A number of refactorings that could be pursued as future work were presented that were obvious design choices after studying the `nhc` source code were presented in Section 7.2. The chapter concluded in Section 7.3 with a extended example for refactoring an expression processing example. In particular a number of the core refactorings from this thesis were used to help extend and design the example from scratch.

Chapter 8

Related Work

This chapter reviews relevant work in the literature in both refactoring and related areas. For an extensive survey of refactoring tools and techniques, Mens produced a refactoring survey in 2004 detailing the most common refactoring tools and practices [80].

The chapter is structured as follows: Section 8.2 explores current research in refactoring including refactoring tools and language-parameterised refactoring. Section 8.3 explores refactoring in a functional programming context. Finally Section 8.4 explores techniques based on the refactoring process such as identifying refactorings from source-code changes.

8.1 Behaviour Preservation and Refactoring

An essential property of a refactoring is that the behaviour of the program being refactored is to be preserved. Ideally, the best approach is to prove that the refactorings preserve the full program semantics. This requires a formal semantics for the program language to be defined. One could argue, however, that with the exception of Standard ML [82], no practical language has a properly defined formal semantics.

Opdyke [86] proposed a set of seven invariants to preserve behaviour of refactorings. These invariants are: unique superclass, distinct class names, distinct member names, inherited member variables not redefined, compatible signatures in member function redefinition, type-safe assignments and semantically equivalent reference and operations. In addition to these, Opdyke's refactorings also included proofs which demonstrated that the enabling conditions preserved the invariants. Tokuda *et al* in [111] also made use of program invariants to show that refactorings preserve behaviour. The notion of a precondition is also used in [118] for formal restructuring using the formal language WSL.

Tip et. al. in [110] explored the use of type constraints to verify the preconditions and to determine the allowable source code modifications for a number of generalisation related refactorings in an object-oriented program language context.

Mens et. al. used a different approach [79] by exploring the idea of using graph transformations to formalise the effect of refactorings and prove behaviour preservation of refactorings. The motivation is that there is a relation between refactoring and graph transformations: a program can be expressed as a graph, and refactoring corresponds to graph transformations. The application of a refactoring is a graph transformation and refactoring pre and post conditions are expressed as application pre and post conditions. [79] also discussed the formalisation of two refactorings: *Encapsulate Field* and *PullupMethod*.

Fowler makes the point about how it is sufficient to write automatic regression tests for refactoring [32]. In his book, he points out that using a test-driven environment is an effective and efficient way of writing stable software systems. In practice, this seems to be the default for many systems, including HaRe [68] and the Erlang Refactoring tool, Wrangler [61].

8.2 Refactoring Tools

A growing number of tools have been developed to automate the refactoring process. In particular, there are a large number of refactoring tools for object-oriented programming languages. Most tools provide the bare essentials of refactorings such as extracting a method, and renaming and moving blocks of code around. Refactoring tools such as Eclipse [28] (described in Section 8.2.4) now offer an extensive range of refactorings also including inlining, extract constant, introduce parameter and encapsulate a field. Many refactoring tools are fully fledged commercial or open-source products. More details regarding the commercial aspect of refactoring engines can be found at Fowler's refactoring website [33].

The following section is a survey of a selection of refactoring tools that are most relevant to the work presented in this thesis.

8.2.1 The Smalltalk Refactoring Browser

The Refactoring Browser [97] was the first successfully implemented tool, and is still one of the most full-featured tools. It supports refactorings for the Smalltalk language [74]. The Refactoring Browser is an extension of the Smalltalk development browser, which offers both program transformation and code browsing facilities.

The Refactoring Browser conforms to Opdyke's preconditions [86] and Roberts' postconditions [98] of a refactoring process.

The Refactoring Browser works by first parsing the code to be refactored into an Abstract Syntax Tree (AST). The available refactorings are coded as templates in the form of ASTs, which may contain template variables. The refactoring is executed by a *parse tree rewriter* that matches the concrete AST with a template AST and performs a tree-to-tree transformation. Finally, the modified AST is passed into the *Formatter* to reconstruct the source back from the tree. Instead of using the standard Smalltalk parser, the Refactoring Browser uses its own Smalltalk parser in order to accept pattern variables and keep comments in the AST. The Refactoring Browser does not support layout preservation.

Refactorings are implemented by using *RefactoryChange* objects in The Refactoring Browser. Each RefactoryChange object also implements an *undo* method which can undo the changes performed by the object. By ensuring that each of the small changes can undo itself reliably, the Refactoring Browser can ensure

Method Refactorings	*Variable Refactorings*
Remove Method	Add Instance Variable
Rename Method	Remove Instance Variable
Add Parameter to Methods	Rename Instance Variable
Remove Parameter from Method	Abstract Instance Variable
Remove Class Method	Create Accessors for Instance Variable
Extract Code as Method	Add Class Variable
Push Up/Down Method	Rename Temporary
Move Method to Component	Rename Class Variable
Inline Call	Inline Temporary
	Abstract class Variable
Class Refactorings	Convert Temporary to Instance Variable
Add Class	Create Accessors for Class Variable
Remove Class	Extract Code as Temporary
Rename Class	Push Up/Down Instance Variable
Convert Superclass to Sibling	Push Up/Down Class Variable
	Convert Instance Variable to Value Holder
	Protect Instance Variable
	Move Temporary to Inner Scope

Table 2: Refactorings implemented in the Refactoring Browser

that complex refactorings can be undone safely, this, in effect, works by keeping a history of small changes. The undo method in the Haskell Refactorer, HaRe also works in the same way.

The refactorings implemented in the Refactoring Browser are shown in Table 2 and they are typical to most object-oriented programming languages. They can be categorized as: *class refactorings* that change the relationships between the classes in the systems, *method refactorings* which change the methods within the system, and *variable refactorings* which change the instances of variables within classes.

8.2.2 CRefactory

Garrido at the University of Illinios at Urbana-Champaign worked on a refactoring tool for C programs called CRefactory [34]. A major challenge with refactoring C programs was that the source code of C programs has preprocessor directives intermixed. Preprocessor directives are hard to handle because directives are lost during transition from the source code to abstract program representations and it is difficult to guarantee correctness in the refactoring. In [35], Garrido proposed an approach to allow incompatible conditional branches of an AST to be analysed and modified simultaneously; this is achieved by maintaining multiple branches in the transformed program tree, each annotated with its respective conditions. In order to include conditional directives in the AST, a pre-transformation phase is performed to ensure that condition directives appear at the same level as external declarations or statements in the C program. As to the implementation architecture, Garrido reused most of the design ideas of the Smalltalk Refactoring Browser [97], with the re-implementation of some components of the architecture in a C context.

8.2.3 Language-parameterised Refactoring

Lämmel proposed the idea of representing program transformations for refactoring in a language-parametric manner using *Strafunski* [64, 65].

The idea of a language-parameterised refactoring (or generic refactoring) framework is this: first, *generic algorithms* are offered to perform simple analysis and transformations in the course of refactoring; second, an *abstraction interface* is provided to deal with the relevant abstractions of a language; and then the actual *generic refactorings* are defined in terms of generic algorithms and against the abstraction interface. The framework is designed in a way that it can be instantiated for different languages, such as Java, PROLOG [12] and Haskell. Lämmel used the *abstraction extraction* refactoring as a running example, and illustrated how this framework can be instantiated for (a subset of) Java. This is a challenging task because of the multiple languages that are subject to analysis and transformation, the program-entity based nature of refactoring tools, the complexity of language semantics and the different semantics between different programming languages. Due to the challenging nature of the work, the project is currently unsuccessful in being integrated into a working application.

Physical Structure	*Class Level*	*Structural*
Rename	Push Down	Inline
Move	Pull Up	Extract Method
Change Method Signature	Extract Interface	Extract Local Variable
Convert Anonymous Class to Nested	Generalise Type	Extract Constant
Convert Nested Type to Tope Level	User Supertype Where Possible	Introduce Parameter
Move Member Type to New File		Introduce Factory
		Encapsulate Field

Table 3: Refactorings implemented for the Eclipse platform

8.2.4 Eclipse

Eclipse [28] is an open source community whose projects are focused on building an extensive program development platform. The Eclipse platform has built-in editing support for Java and Erlang, amongst many other programming languages. Aside from just an editor, Eclipse also provides a refactoring environment for Java programs built on the Java Development Tooling platform [29]. Eclipse includes basic refactorings such as safe *rename* and *move* refactorings, advanced refactorings such as *Extract Method* and *Extract Superclass* together with complex refactorings which can be applied across large projects such as *Use Supertype* and *Infer Type Arguments*. The Eclipse refactoring framework also includes an API to allow programmers to devise their own refactorings. The Eclipse platform contains three types of refactorings: refactorings that change the *physical structure* of the code and classes; refactorings that change the code structure at the *class level*; and refactorings that change the code within a class at the *structural* level. Table 3 shows the current refactorings implemented for the Eclipse Ganymede platform.

8.2.5 NetBeans

NetBeans [30] is an IDE that supports —amongst others— Java, C, C++ and JavaScript. Similar to Eclipse (see Section 8.2.4), NetBeans also provides a refactoring framework and API for Java developers [31]. Starting from NetBeans IDE 4.0, a new Java language infrastructure and refactoring API framework needed for implementing refactorings has been devised. So far the NetBeans development team have implemented some basic refactorings as a proof of concept. Future releases of NetBeans will contain a more extensive list of refactoring features. Table 4 shows the current refactorings that are supported by the NetBeans 6.1 framework.

8.2.6 Eclipse vs. NetBeans

Eclipse and NetBeans both claim to be universal tool platforms. NetBeans, however, lacks an open-source compiler and an AST model for the development of refactorings; the open-source compiler and AST model will impede the development of refactorings for vital Java extensions and structure-aware support. Lack of structure-aware support means that NetBeans falls short for extensions of Java such as AspectJ [56].

The Eclipse platform, on the other hand, has tight integration for advanced features such as refactoring, inheritance and call hierarchy views, structured search and structure-aware repository report. Unlike NetBeans, these features are not plug-ins: they are at the heart of the Eclipse platform. This also makes it possible to extend these features without having to re-implement basic strategies such as AST rewriting and traversal.

Class Refactorings	*Method Refactorings*	*Variable Refactorings*
Extract Interface/Superclass	Pull Up Method	Safe Delete
Convert Anonymous to Inner Class	Push Down Method	Move Inner To Outer Level
Use Supertype Where Possible	Extract Method	Rename Instance Variable
Move Class		
Rename Class	Rename Method	Rename Variable
	Change Method Signature	Extract Local Variable
	Encapsulate Field	Inline Local Variable
	Inline Method	
	Move Method	
	Generify	
	Clean-Up Code	
	Replace Method Duplicates	

Table 4: Refactorings implemented in NetBeans

8.3 Refactoring in Functional Languages

Program transformation for functional programs has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [88]. Other work in program transformation for functional languages is described by Hudak in his survey [45]. In a general sense, a program transformation is a conversion of one source program into another. In the case of a refactoring, and correctness preserving transformations, the transformation is *bi-directional*, as a refactoring does not alter the program's semantics.

8.3.1 Refactoring Erlang

The University of Kent and Eötvös Loránd University are now in the process of building a refactoring tool for Erlang programs [61]. However, different techniques have been used to represent and manipulate the program under refactoring. The Kent approach uses the *Annotated Abstract Syntax Tree* (AAST) as the internal representation of an Erlang program, and program analyses and transformations manipulate the AAST directly. The Eötvös Loránd approach uses the relational database MySQL [81] to store both syntactic and semantic information of the Erlang program under refactoring; therefore, program analyses and transformations are carried out by manipulating the information stored in the database. The rest of this sub-section describes the two approaches, and compares them.

The Kent Approach

In this approach, the refactoring engine is built on top of the infrastructure provided by SyntaxTools [16]. Wrangler uses the *Annotated Abstract Syntax Tree* (AAST) as the internal representation for an Erlang program. The SyntaxTools library contains modules for handling Erlang Abstract Syntax Trees (ASTs), in a way that is the analogue of the methods described for HaRe in Chapter 2:

- The uniform representation of the AST nodes and the type information preserved by the AST allow the Kent group to write generic functions that traverse into subtrees of an AST while treating most nodes in a uniform way, and treating the nodes with a specific type in a specific way.

- SyntaxTools allows comments to be preserved in the AST, this liberates the Kent group from the problem of preserving comments in the AST.
- The AST contains the following information: *binding* information, which allows for, say, the checking of whether two variable names may refer to the same entity; *location* information, which makes it easier to map a syntax phrase selected from its textual representation in the editor to its AST representation; *category* information, which allows entities such as patterns to be distinguishable from expressions; and *type* information, which is currently a work in progress.

The Eötvös Loránd Approach

Instead of annotating ASTs with the information that is necessary for program analysis and transformation, the Eötvös Loránd approach uses a relational MySQL database; the database is used to store both abstract Erlang syntax trees and the associated static semantic information. SQL is then used to manipulate the stored database information.

A number of database tables are used to store different information about the program under refactoring: the syntax-related table corresponds to the *node types* of the abstract syntax tree of Erlang as introduced in the Erlang parser. Semantic information, namely location and binding information, is stored separately in an extensible group of tables. Adding a new feature to the refactoring tools therefore requires the implementation of additional tables storing collected semantic information.

Consider the code fragment in Figure 47 and its AST database representation in Table 5; the table elements: *clause*, *name*, *infix_expr* and *application* refer to the corresponding syntactic catergories. Without addressing any further technical details, one may observe that each table relates parent nodes of the corresponding

$$\texttt{gcd}_{30}\texttt{(N}_{15}\texttt{, M}_{16}\texttt{) when N}_{17}\texttt{ >=}_{18}\texttt{ M}_{19}\texttt{ ->}$$
$$\texttt{gcd}_{23}\texttt{ (N}_{24}\texttt{ -}_{15}\texttt{ M}_{26}\texttt{, M}_{28}\texttt{);}$$

Figure 47: Source code of the example function clause

information in the AST	database equivalent	
	table name	record in that table
1st parameter of clause 30 node 15	clause	30, 0, 1, 15
the name of variable 15 is `N`	name	15, "n"
2nd parameter of clause 30 is node 16	clause	30, 0, 2, 16
clause 30 has a guard, node 22	clause	30, 1, 1, 22
the left and right operands and the operator of the infix expression 20 are nodes 17, 19 and 18, respectively	infix_expr	20, 17, 18, 19
the body of clause 30 is node 29	clause	30, 2, 1, 29
application 29 applies node 23	application	29, 0, 23
the content of atom 23 is `gcd`	name	23, "`gcd`"
1st param. of application 29 is node 27	application	29, 1, 27

Table 5: The representation of the code in Figure 47 in a database

type with their child nodes.

A Comparison Between the Two Approaches

The major difference between the two approaches discussed lies in how the syntactic and semantic information is stored and manipulated. The Eötvös Loránd approach needs more time and effort on database design and migration of information from Erlang abstract syntax trees to the database; whereas the Kent approach is relatively easier to maintain and store information about refactorings. However, as the Eötvös Loránd approach tries to avoid reconstruction of the database between two consecutive refactorings —by incrementally updating the database, so as to keep the stored syntactic and semantic information up-to-date— it may be worth the effort. The two universities would like to test the two approaches further, once both tools support a more substantial number of representative, module-aware refactorings [69].

8.3.2 The Haskell Equational Reasoning Assistant

The Haskell Equational Reasoning Assistant, HERA [36] is a system that provides both a GUI level and a batch level Haskell rewrite engine inside a single tool. The interactive interface is used to create and edit non-trivial translations that can be used to extend the batch level API. The batch level API can be used to implement powerful, context sensitive rewrites that can be provided to the interactive interface.

HERA uses the Haskell AST provided by Template Haskell [94], and provides three mechanisms for specifying rewrites over the AST. The rewrite strategies are primitive rewrites, such as beta-reduction and case rewrite rules as provided in the Haskell report [92]. The rewrites are implemented as AST to AST transformations.

HERA records every rewrite as an *equation*, with the left hand side as the starting fragment, and the right hand side as the fragment after all the rewrites are applied. Any equation written using HERA stores a list of primitive rewrites or equations applied, allowing for easy playback, basic proof management and presentation opportunities.

HERA shares the basic properties of HaRe. It is important to notice a difference however, HaRe works purely at the source level of a program, and applies well-understood software engineering patterns. HERA handles large-scale rewrites in a different way, using only a series of small steps performed in a strict bottom up manner. The HERA user focuses on one binding group at a time, eventually splitting the rewritten functions into a series of workers and wrappers [38]. The wrappers respect the original interface of the function, and the workers represent the rewritten function. Inlining any newly created wrapper inside any call side is correctness preserving. It is possible to implement particular refactorings from HaRe in HERA such as renaming and generalisation. However, the HERA tool doesn't provide an advanced API for program transformation and so refactorings would have to be described in terms of small transformations, which in some respects would make it more difficult to scale to large-scale transformations.

8.3.3 Kansas University Rewrite Engine

The Kansas University Rewrite Engine (or KURE) [37] is a Haskell hosted Domain Specific Language (DSL) for writing transformation systems based on rewrite strategies. KURE is hosted in Haskell and provides a small set of combinators that can be used to build parameterized term rewriting and general user-defined rule applications. KURE builds upon the tradition of systems like Stratego [115] and Strafunski [64] by adding a strongly typed strategy control language. It is intended for writing reasonably efficient rewrite systems, makes use of type families to provide a delimited generic mechanism for tree rewriting, and provides support for efficient identity rewrite detection.

KURE is an attempt to revisit the design decisions of Strafunski and Stratego, and to build a more efficient rewrite system hosted in Haskell. Specifically, KURE replaces the powerful hammer of "scrap your boilerplate" generics [53] provided in Strafunski with a more precise, user configurable and lightweight generics mechanism. KURE was used (along with Template Haskell) to provide the basic rewrite abilities inside HERA.

8.3.4 Fold/Unfold

The fold/unfold system of Burstall and Darlington [14] was intended to transform recursively defined functions. The overall aim of the fold/unfold system was to help programmers to write correct programs which are easy to modify. There are six basic transformation rules that the system is based on:

- *Unfolding*: Replacing a function call with a body of the function where the actual arguments are substituted for formal arguments.

- *Folding*: Replacing an expression with a call to a function if the function's body can be instantiated to the given expression with suitable arguments.

- *Instantiation*: Introducing a substitution instance of an existing equation.
- *Abstraction*: Introducing a where clause by assigning a new name to an expression. This can be used in combination with folding and instantiation to introduce sharing, and hence gain efficiency.
- *Definition*: The addition of a new function declaration.
- *Laws*: The use of laws about the primitives of the language.

The advantage of using this methodology is that it is simple and very effective at a wide range of program transformations which aim to develop more efficient definitions; the disadvantage is that the use of the *fold* rule may result in non-terminating definitions.

8.3.5 The Munich CIP Project

Another representative work of program transformation was the Munich CIP (Computer-aided Intuition-guided Programming) [8, 7]. The central idea of this project was to develop programs by a series of small, well understood transformations. CIP uses a rule-based language to describe transformations, and all transformation rules are specified by laws about program schemes [89]. For instance, a rule to eliminate a condition statement can be expressed as:

$$\frac{\texttt{if}\ B\ \texttt{then}\ E\ \texttt{else}\ E \qquad DEF(B)}{E}$$

This rule uses scheme variables B and E to identify parts of the expression to be transformed. The other condition of this rule states that the transformation is only valid if the boolean expression B is defined.

Somehow similar to refactoring tools, the expressiveness of this approach depends on the available transformation rules, so systems using this approach may have dozens of transformation rules, and the user needs to search for applicable rules to solve the problem at hand.

8.3.6 The Bird-Meertens Formalism (BMF)

The Bird-Meertens Formalism (BMF) [5], also called Squiggol, is a calculus for deriving programs from their specification by a process of equational reasoning. It consists of a set of higher-order functions that operate mainly on lists including map, fold, scan, filter, inits, tails, cross product and function composition. BMF is based on a computational model of categorical data types and their accompanying operations. Program developments in BMF are directed by considerations of *data* structure, as opposed to *program* structure.

8.3.7 Other Program Transformation Systems

Other recent work on program transformation aimed for program derivation and optimisation includes:

- The Ultra [39] system which can assist programmers in deriving correct and efficient Haskell programs formally and interactively. The design of Ultra is conceptually based on CIP-S [7], but with some modifications.

- PATH (Programmer Assistant for Transforming Haskell) [112] is another interactive program transformation system. The system is aimed to leverage the advantages of both the fold/unfold approach and that taken by the CIP project, and the disadvantages of neither.

- The HsOpt optimiser [116] for the Helium compiler, a subset of Haskell developed at Utrecht University, in the transformation language Stratego.

- The prototype framework HSX [117] for experimentation with the application of rewriting strategies using the transformation language Stratego for program optimisation.

- The MAG program transformation system for a subset of Haskell, developed by O. de Moor and G. Sittampalam [23].

Program derivation and optimisation within the functional paradigm has been studied extensively. However, they pose problems for a refactorer. Program derivation transforms simple, obviously correct, programs into more efficient and usually more complex variants. Transformations for program optimisation within compilers, (e.g. deforestation [106]), which is used to eliminate intermediate trees tend to be localised, addressing a program's control or data flow and are hidden from the user. The kind of program structure considered for refactoring is often non-localised and related to the overall program design and knowledge representation – they are more commonly related to large scale declarative aspects rather than smaller scale operational ones.

8.4 Refactoring Support

There has been a great interest in refactoring-frameworks, such as automatically detecting where refactorings can be applied in a source program and how refactorings could be composed together. This section discusses how refactoring tools can be taken a step further with tools and techniques to help the programmer in designing and applying refactorings.

8.4.1 JunGL

Jungle Graph Language (JunGL) is a domain-specific language developed by Verbaere et. al. [114] for refactoring and is designed to enable developers and tool authors to implement, easily and correctly, refactoring support for any language. JunGL is, however, principally aimed at object-oriented paradigms.

JunGL is a hybrid of a functional-based language in the style of ML and a logic query language similar to Datalog [3]. JunGL manipulates a graph data structure that represents all information about a program including ASTs, variable binding, control flow and dataflow. JunGL has a notion of demand-driven evaluation for constructing edges of the graph, and allows powerful path queries to be run over the program. Like ML, JunGL has features such as pattern matching and higher-order functions, while also allowing the use of updatable references (as does ML).

The main data structure manipulated by functions in JunGL is a graph that represents the program one wishes to transform. Each node and edge in the graph has a *kind* that is represented by a string. For example, edges that indicate the control flow from one statement to the other are labeled "csfsucc", for *Control Flow Successor.*

The graph will consist of only ASTs initially —where the edges indicate children and parents— additional computed information is added via "lazy" edge definitions. These definitions will only be evaluated when their value is accessed.

The object query graph (similar to the AST) is manipulated by a number of queries to find out specific information where destructive updates may be performed on the graph. The notion of a predicate was added to the JunGL language, this effectively makes JunGL a hybrid functional-logic language.

Predicates are most commonly created by constructing *path queries*: regular expressions that identify paths in the program graph. For example, here is a predicate that describes a path from a variable occurrence, *var*, to its declaration:

```
[var]
parent+
[?m:Kind("MethodDecl")]
child
[?dec:Kind("ParamDecl")]
&
?dec.name == var.name
```

The path components between square brackets are conditions on nodes, whereas *child* and *parent* match edge labels. The above example embodies the requirement that we can reach a method declaration by following one or more *parent* edges from the *var* node. Furthermore, that method declaration (named *?m*) has a child that is a parameter declaration (named *?dec*). Finally, the last conjunct checks that the parameter's name coincides with the name of the variable node we started out with. It is important that the variable occurrence *var* is known, so that the predicate above can be used to find instantiations of the logical variables *?m* and *?dec* so that the above predicate is true.

The design of JunGL was validated through a number of non-trivial refactoring scripts on a substantial subset of the C# language [73]. In particular, some bugs in the IntelliJ IDEA [11] and Visual Studio system were demonstrated and discussed by expressing the refactoring in JunGL.

8.4.2 Guru

Guru [83] is a prototype tool developed by Moore for restructuring inheritance hierarchies expressed in the Self programming language [113]. Although not a refactoring tool, Guru can automatically restructure an inheritance hierarchy into an optimal hierarchy for all the objects currently in the system, whilst preserving the behaviour of programs. Optimal means that there are no duplicated methods and there are the minimum of objects and inheritance relationships required for such an inheritance hierarchy.

The optimisation is achieved by first creating a copy of the objects to be restructured and discarding the inheritance hierarchy, then building a replacement inheritance which ensures no duplication.

Moore found that the inheritance hierarchies produced by Guru are easy to understand when restructuring code that is easy to understand. For code that is not easy to understand, the inheritance hierarchies created by Guru may bear so little resemblance either to the original system, or to any concepts that are understood by the programmer or designer, so the restructured system may be difficult to understand, although it can assist the programmer in identifying the faults in the original design.

8.4.3 Star Diagram

Developed by Bowbridge [10], the Star Diagram is a graphical visualisation tool, providing a hierarchical tree-structured representation of the source code relating to a particular data type, eliding code unrelated to the data structure's use. Similar code fragments are merged into *node stacks* to reveal potentially redundant computations. The tool can help the programmer design transformations based on how the transformation is distributed and how the fragments are related to each other. The visualisation is mapped directly to the program text, therefore manipulation of the visualisation also restructures the program.

The visualisation provided by the Star Diagram system is targeted at supporting the specific task of data encapsulation. Other kinds of transformation would require the assistance of other views.

Tools based on the notion of Star Diagram have been developed for C, Ada and Tcl/Tk [24]. Korman applied the Star Diagram concept to Java programs and implemented a tool called *Elbereth*. In [60], he described how programmers can be supported in performing a variety of refactoring tasks, such as extracting a method or replacing an existing class with an enhanced version. While the tool can assist the programmer in *planning* the restructuring, the restructuring itself has to be performed by hand.

8.4.4 CatchUp!

In [42], Henkel and Diwan present an idea to record how a library developer changes a software project, or API. As the library developer invokes particular refactorings over the library, CatchUp! records them. The refactoring changes are then replayed back to update any client code automatically. This allows recording and replaying of software refactorings to be performed with little cost for both the library developer and the client maintainer. The approach is very lightweight in that it does not require a central version repository system or a configuration management system; the approach also integrates well with modern development environments, such as Eclipse [28].

Deprecation in Java implementations (and, indeed, in any other language implementation) points out serious problems with library development: once a particular API is developed, developers are forced to maintain it. Developers may choose to deprecate and later remove unwanted methods and types, but this largely depends on the amount of client code that needs to be fixed in order to migrate to a new API. CatchUp! allows this transition to be easier by applying a refactoring automatically to remove deprecated entities.

CatchUp! allows refactorings to be applied automatically to client programs by means of the programmer choosing the recorded refactorings from a list. Since subsequent refactorings can destroy the results produced by previous refactorings, CatchUp! uses a step-by-step view showing each stage of the refactoring process. This is particularly useful if the developer doesn't wish to apply all the recorded refactorings.

The strategy used by CatchUp! is to extract enough information from the refactorings so that they can later be replayed. To replay refactorings, CatchUp! recreates Eclipse's refactoring objects according to the specifications given in the refactoring trace. Currently, CatchUp! supports a limited number of Java refactorings that are embedded within Eclipse, with support to script these refactorings by recording the application of the refactorings over an API; the recorded scripts are then executed over a client code base.

8.4.5 Static Composition of Refactorings

In [57] Kniesel argues that the lack of user-definable refactorings is equally unsatisfactory for tool providers and for their users: for tool providers who continuously invest time and money in the infinite evolution of refactorings; for the users, because they are forced either to wait for some release, hoping to find their desired refactoring there, or to implement their own refactorings. However, the latter option is probably not ideal to most users, as implementing a lot of time and effort into a new refactoring is probably not worth it, especially if they have a tight schedule to develop a project.

A solution to this problem is a provision to allow users to compose their own refactorings together using some kind of higher order language that makes it easy to describe compositions. Kniesel describes in his paper a method for a static composition of refactorings. In his paper he introduces a *refactoring editor* that essentially allows one to create, compose and edit refactorings.

Although the main motivation for this work and its primary application is in the domain of refactorings, the paper takes a much broader approach. The paper describes a *conditional transformation* as their basic concept. A condition transformation (CT) is a pair consisting of a precondition and its transformation, where its transformation is performed on a given program only if its precondition evaluates to true. The following discusses the various approaches of composition that the paper investigated:

- **Composition:** in order to compose transformations, the preconditions of each of the transformations in a given composition must be analysed. Each transformation in the chain must satisfy the preconditions of the next transformation in the chain.

- **AND-sequences:** in many transformation scenarios the execution should not continue if one CT in a sequence fails. It is also often the case that each transformation in a given chain is applied. Execution of an AND sequence of conditional transformations applies *each* transformation, provided that *each* precondition is true when it is checked. Otherwise, none of the transformations are applied.

- **OR-sequences:** an OR-sequence is the simplest form of a sequence in that the preconditions of the individual CTs are not related in any way.

- **Enabling Refactorings:** The composition can infer that earlier refactorings can set up the preconditions of later ones. This is particularly useful when certain preconditions cannot be evaluated by static program analysis.

- **Chaining Undo:** in a conjunction of refactorings, it is useful to include the possibility that something will go wrong. An undo function to roll back to before the composition was applied is necessary, as in as undo to roll back through each step of the composition.

Conditions for a refactoring via a transformation description; each refactoring states whether a certain condition must hold or not to succeed. For each program transformation one can describe effects on conditions either *forward* or *backward*. A *forward description* of a transformation T takes a condition that holds (immediately) before the transformation is performed and returns the condition that will hold (immediately) afterwards. A *backward description* of a transformation T takes a condition that is assumed to hold immediately after the completion of T and produces the equivalent condition to be checked before performing T.

In his pioneering work, Opdyke [86] introduced many of the ideas on which much of the current work on refactorings is based. This includes the use of preconditions for ensuring behavior-preservation, their evaluation on the basis of a set of analysis functions and the idea of composing transformations. However, his notion of composition is sequential execution, corresponding to Kniesel's notion of OR-sequence.

The Smalltalk Refactoring Browser [97] has postconditions that correspond conceptually to Kniesel's forward transformation descriptions. Analysis functions and conditions can be regarded as abstractions of programs. Their transformations reflect the semantics of a program transformation at a more abstract level. Kniesel's approach eliminates the intermediate level of analysis functions, describing the effect of transformations directly on conditions. Instead of transforming interpretations of function and predicate symbols with respect to a program, transformation descriptions simply rewrite predicate symbols.

8.4.6 Munich CIP-S

CIP-S [7] provides an extension to the Munich CIP system by allowing the manipulation of concrete programs. CIP-S also supports the derivation of new transformation rules, transformation of algebraic types and the verification of the applicability conditions.

The system includes a generative set of transformations, including general principles (such as fold, unfold, etc. [14]), the definitional rules of the language and the axioms and inference rules for the predefined types. In addition to these, some frequently used derived rules about language constructs and a few general techniques concerning control constructs are also made available.

These default rules provide the user with a basis for which to derive further rules that they consider useful for the particular problem domain that they are working on. The CIP-S system guarantees that all rules that are inferred within the system are correct with respect to the initial collection of rules. Rules that are not inferred in this way are left to the responsibility of the user to ensure their correctness.

The default rules in the system are *schematic* rules over which more complex rules can be derived as transformation algorithms. These transformation algorithms serve for partly automating the transformation process.

The core of CIP-S is independent of a particular language, instead special variants of the system are derived for any algebraically defined language. This also means that other languages that are used by the system (i.e. for operating the system and for formulating transformational tasks) are also exchangeable as well. The standard implementation of CIP-S uses the language of CIP-L [8] in its Algol or Pascal variant.

CIP-S was designed to be used as an extensive software development tool. Hence it supports each individual step and task that may occur within the program development cycle. The starting point for a formal development usually consists of a formal specification; the CIP-S system can aid the formulation of a formal specification by offering the following facilities:

- A basic catalogue of the standard data-types that are frequently used at the specification level.
- Various *surveys* over these types, including:
  - the set of types;
  - the laws of individual operators;
  - the relations among the types.
- Support for the modification of already available types, including enrichment, restriction, renaming and parametrisation.
- Support for the instantiation of a type, together with the instantiation of the transformation rules gained from its laws.

This system support facilitates the formulation of specifications as the necessary data-types are already available or obtainable by the modification of existing data types. In addition to these facilities, CIP-S also provides the following features with respect to the transformation system:

- Provides the basic catalogue of schematic transformation rules that express general programming techniques;
- Search facility for syntactically applicable rules;
- A graphical representation of rule applications showing the instantiation gained by the matching process (and showing the variables not instantiated).

The CIP-S also keeps a history of the transformation steps allows the use to backtrace to previous steps if necessary.

8.5 Summary

The main challenges faced by automating the refactoring process are: a) identifying refactoring opportunities, b) global behaviour-preserving program transformation, and c) program appearance preservation. Automating the refactoring process benefits from the previous work on program lexing, parsing, analysis and transformation, but also exposes new research areas, such as producing ASTs or token streams with richer information, verifying global behaviour preservation, a general framework for building refactoring tools etc. As functional programming languages and object-oriented programming languages expose different program structures, it is no surprise that each of them have a collection of refactorings particularly appropriate to their own program structures.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This thesis has explored transforming (in the sense of both general program transformation, and refactoring) and analysing the functional programming language Haskell. The thesis has extended the existing work on refactoring Haskell programs made by Li in her PhD thesis [67] and has also aimed to complement the existing work in program refactoring and transformation in other functional programming and object-oriented fields. Specifically, the contributions of this thesis are as follows:

- The creation and study of a set of refactorings for Haskell (structural refactorings were described in Chapter 4 and data type based refactorings were described in Chapter 5). A collection of Haskell refactorings have been derived and analysed in terms of their side-conditions and transformation rules. Side-conditions together with the transformation rules guarantee that a refactoring does not change the program's external behavior.

- Enhancing the design and implementation of HaRe and its API. This thesis introduces a collection of new refactorings and transformations for Haskell. The API for HaRe is extended to include additional functionality for these new refactorings. In particular, additional type-checking query and transformation functions were added to the API.

- The investigation and implementation of duplicate code elimination for Haskell (described in Chapter 6). Using the infrastructure of HaRe, a set of refactoring and analyses have been developed to aid the programmer in removing duplicate code. A survey comparing the duplicate code analyser developed as part of this thesis with duplicate code analysers for other languages is undertaken. The clone analysis and extraction is also extended to take sequences of monadic "commands" into account, as well as common substitution instances of expressions.

- Program slicing (described in Chapter 3). The investigation and implementation of program slicing techniques as Haskell transformations to help the programmer remove redundant code. In particular, function *splitting* and *merging*. *Splitting* is concerned with taking a function returning a tuple and creating new functions encapsulating the semantics of each tuple element (or from a subset of fields from the original tuple). The converse, *merging*, is taking a set of functions and fusing them together, forming a new function returning a tuple.

- The investigation of and inclusion in HaRe of the type checking facilities of the Glasgow Haskell Compiler (GHC) [77] (expansion on the type checking is discussed in Section 5.1). The type checker can be used to add additional side-conditions and also to act as a partial control for the post-conditions: checking that a program is still type-safe after a particular transformation. For example, as part of the *splitting* process, the Haskell module is type checked. If the type-checking fails, the refactoring is not performed.

- A case study into refactoring Haskell programs (described in Chapter 7). The case study not only looks at current refactorings for HaRe, but also discusses potential refactorings that could be added to HaRe to aid better understanding of the program; some of these refactorings are discussed in more detail in future work.

A particular limitation lies with designing refactorings in their full generality instead of a large set of smaller, simpler refactorings. Simpler refactorings can be described clearly, with a clear set of conditions and limitations. Larger refactorings, that aim to be more general, and which are designed and implemented for this thesis, are difficult to describe in terms of both the conditions and the limitations. A possible solution to this would be to formalize the refactorings, but then a danger lies in over-complicating an already complex issue.

Haskell is a very complex language, and its model of type signatures, pattern matching, guards, where clauses and type classes, makes it a difficult language to refactor. Indeed, for many of the refactorings discussed in this thesis, there have been occasions where it is not clear what the refactoring should do. In most cases, it was decided to take a tractable subset of the full diversity and provide a full solution to a subset rather than a partial solution to all.

For example, when adding a data constructor as discussed in Chapter 5, it is possible to come across situations where we have a recursive data structure defined and pattern matching over that data structure:

```
data T = C1 T  | C2


f :: T -> Int
f (C1 (C1 C2)) = 3
f (C1 C2)      = 2
```

Adding a new constructor, `C3`, to the type, `T`, creates a situation where it is not exactly clear what new equations must be added for `f`. Should we create new equations for `f` introducing a new pattern match for the new constructor where it could occur in the previous equations?

```
data T = C1 T  | C2 | C3
addedC3 = error "Added C3 to T"
```

```
f :: T -> Int
f (C1 (C1 C2)) = 3
f (C1 (C1 C3)) = addedC3
f (C1 C2)      = 2
f (C1 C3)      = addedC3
f C3           = addedC3
```

Or only one new equation, which captures the simplest of situations?

```
data T = C1 T  | C2 | C3
addedC3 = error Added C3 to T"
f :: T -> Int
f (C1 (C1 C2)) = 3
f (C1 C2)      = 2
f C3           = addedC3
```

The refactorings in this thesis have tried to follow the simplest design choice as possible so not to refactor code into something that the programmer does not expect. The solutions for adding a new constructor, therefore, adds one new equation so at least calls to `f` will not result in a pattern match error if `f` is called with the value `C3` (and, more importantly, gives an error indicating that `f` is not defined for the newly added constructor `C3`). It may be noted, however, that some users would consider the first example to be correct, as it covers all situations where the previous equations could be called with the value `C3`.

The case study presented in Chapter 7 resulted in some interesting conclusions. It seems that the simpler, atomic, refactorings are more useful in refactoring large-scale programs than the larger more complex ones. Some of the more complex refactorings defined in this thesis may only be applicable in specialized situations. This certainly gives motivation to create larger refactorings from lots of smaller atomic refactorings. The most commonly used refactorings were *introduce new definition*, *generalise definition* and *folding*; these three refactorings exploit the idea of higher-order polymorphism resulting in abstracting code [107].

In addition, the case study also begged the question as to whether it would have been more appropriate to have performed the case-study first as part of the design of HaRe, and then to have proceeded with designing refactorings that are obviously necessary. However, researching extensively into refactoring for Haskell first, and then performing the case study at the end allows the opportunity for more refactoring cases to be observed, as designing and implementing refactoring helps one to understand their implication and use.

The view that has been taken for this thesis is that of implementing atomic operations from which more complex refactorings can be constructed. However, in hindsight it would have been more useful to separate out the atomic components of the refactorings so that they could be executed singularly if desired.

A general conclusion from designing large-scale refactorings for Haskell is the lack of viable refactoring tool-support. It seems that rather than spending time and effort in designing and implementing a large refactoring set for Haskell, more time has been invested in learning and understanding the platform architecture on which HaRe sits.

9.2 Future Work

The work presented in this thesis can still be carried forward in a number of directions.

- Adding more refactorings to HaRe. The number of refactorings for HaRe has increased, but there are still a number of refactorings listed in the catalogue [96] that are still awaiting implementation. An example of a few of the more useful structural refactorings are given here.
    - **Swap Arguments**. Swap the position of two arguments of a function, this requires all argument lists to be transformed to take the swapped arguments into account. It may be better to implement a separate *permute arguments* refactoring (or define *permute arguments* as a series

of swaps), and then make this a more special case.

- **(Un)Currying** of function arguments. A subsequence of n arguments of a function is selected for conversion to an n-tuple. This refactoring will require the conversion of all argument lists, type signatures and uncurrying must also take partial applications into account.
- **Replacing a multi-equation definition** with a single-equation definition using `case` expressions. This transformation has already been defined, to some extent by case analysis introduction (in Section 5.6.3) and by *unfolding*, but is certainly worth being a separate, more general, refactoring. This refactoring would make use of *uncurrying*, alternatively a series of nested `case` statements, one per argument, will be required.

• Chapter 5 discusses various type-level refactorings; specifically, refactorings that transform at a type level rather than at a term level. There are still a number of refactorings that could be implemented using the type checking facilities derived for the work in Chapter 5. In particular, these might include:

  - **Name a type**. Identify uses of a type in a particular way by making them instances of a type synonym. This refactoring was identified as an important future refactoring in the case study in Section 7.2 on Page 169. This has no semantic effect on the program except that type synonyms cannot be the subject of any `instance` declarations within the program. This refactoring could be used to remove duplicated code at the type level and to also better understand a Haskell project.
  - **Introduce `class` and `instance`**. This can be done by identifying a type and a collection of functions over that type. This trades off pattern matching (on the algebraic type) against extensibility, à la OOP and will not work well with binary methods. This can be seen as a sequence of simpler refactorings, including splitting the sum into its constituent

parts, introducing an interface (class) for the functions over the types and so forth.

```
data Shape = Circle Float | Rect Float Float
area (Circle r) = pi * r^2
area (Rect h w) = h * w
```

It is possible to introduce an existential quantification over `Shape` and then have overloaded implementations of `area` over `Shape`.

```
data Shape = forall a. Sh a => Shape a
class Sh a where
   area :: a -> Float

data Circle = Circle Float

instance Sh Circle
 area (Circle r) = pi * r ^2

data Rect = Rect Float Float

instance Sh Rect
  area (Rect h w) = h * w
```

This approach is not without its drawbacks. In Haskell 98, binary operators become problematic. For example it is possible to define `(==)` on the first version of shape using the standard definition by case analyses. Instead it is necessary to convert shapes to a single type (via `Show`) to turn a case analysis over types into corresponding cases over values. Unfolding could also be extended to take the inlining of overloaded functions into account. Binary operators are not just an issue for Haskell classes, they are a well recognised problem in OO in general. Consider the following:

```
class MyInt
{
   private:
      Word32 i;

   public:
      MyInt add (MyInt other);
}
```

In the above, `add` has the type `MyInt -> MyInt -> MyInt` as it is encoded as taking an implicit parameter, in this case the class that it is defined in. If we were to add a subclass `MyInt2`:

```
class MyInt2 (MyInt)
{
   public:
     MyInt2 add (MyInt2 other);
}
```

The overloading properties of OO programming prevent us from changing the type of `add`, so `MyInt2` would be disallowed. Consequently, for `MyInt2` this would force `add` to have the meaningless type `MyInt2 -> MyInt -> MyInt` for instances of the `MyInt2` class.

- Make more use of type information with the current refactorings in HaRe. For instance, when generalising a function definition that has a type signature declared, the type of the identified expression needs to be inferred, and added to the type signature as the type of the function's first argument. Converting a pattern binding in a `where` to a pattern binding in a `let` may make an originally polymorphic definition monomorphic, and fail at program compilation. Adding a correct type signature to the lifted pattern binding would solve this problem.

- Coping with literate Haskell modules. The current Programatica-based HaRe only refactors source modules that are not literate Haskell files. As some users —particularly those who are learning Haskell— use the literate style of Haskell programming, it would be beneficial to allow HaRe to cope with literate files, albeit this is a minor issue.

- Coping with Haskell pre-processing directives. Currently the Programatica-based HaRe does not pre-process pragmas in a Haskell program. In particular, this occurred when refactoring nhc [100] in Chapter 7, as the source modules had to be run through `cpp` [49] first. This problem would also relate specifically to a tool designed to refactor C and C++ programs: as the use of pre-processing pragmas (including the use of `#include` directives) are an essential part of the C and C++ programming languages. It is therefore plausible to port solutions for refactorings tools for C and C++ over to HaRe to attempt to solve the pre-processing issue.

- Moving HaRe to a more advanced compiler API. Currently, HaRe works with the Haskell 98 standard of Haskell. However, most Haskell users currently work with the GHC standard of Haskell. The Programatica compiler is currently not being maintained and as Haskell is moving more towards the Haskell Prime standard [21] it makes even more sense to move compiler front-ends. There has been some work by Ryder [102] on porting HaRe over to the GHC API; currently, Chaddaï Fouché is working towards porting a basic refactoring of HaRe over to GHC.

- In the future, it is hoped that HaRe will be extended to allow refactorings to be scripted. Scripting refactorings allows elementary —or atomic— refactorings to be stitched together, creating the effect of a complete refactoring process. For example, with the folding refactoring described in section 4.1, it is possible to define the refactoring in terms of *if..then..else to guards* and *folding* rather than the two refactorings being part of a more complex one. For the introduction of pattern matches, as described in Section 5.6.1, it is also possible to define the refactoring with *introduce sub-pattern*, *unfolding* and *case analysis simplification* (indeed, an example is given of this in Section 4.7).

The more atomic refactorings that HaRe provides, the more cases there are for refactoring composition and the more likely there will be some commonly used refactoring patterns. It would be helpful, therefore, to have a simple script language that can be used by HaRe or the users to easily build composite refactorings from existing ones. The script language should be expressive enough to contain constructs that allow sequencing, iteration and alternation to be defined. The script would also need to be interpreted by HaRe and stored, so that the script can be invoked once it has been defined.

An immediate problem that comes from scripting refactorings in HaRe is how to retain, or define, the parameters for the sequenced refactorings. Usually, in HaRe each refactoring takes a program entity of interest as a parameter (usually as a start and end location, or as a region specified by start and end locations). In scripted refactorings, the user will not have access to the intermediate results of the refactorings, and will therefore not be able to highlight any code entities for parameterising the intermediate refactorings. In addition to this, the result of one refactoring may transform the program in such a way that identifiers change from refactoring to refactoring, this may prevent the user from identifying the changes and inputting new parameters for the remaining refactorings in the composite chain. It may only be possible, therefore, to script refactorings that share a similar set of arguments so that the parameter problem can be solved with AST information. This problem could be solved using addressing somehow, which is rather more semantic than the current "position in a text file" that HaRe currently uses.

A naïve way of executing a composite refactoring is to execute the involved atomic refactorings in the specified order; the composite refactoring fails if one of the atomic refactorings fail, and succeeds if all the atomic refactorings succeed. It is not always possible, however, to take the union of side-conditions for the atomic refactorings to derive the side-conditions for the composite refactoring. Some conditions that are not satisfied by the current program may become satisfied after certain refactorings are performed, and vice versa. Post-conditions were introduced as a means to determine the side conditions for composite refactorings [98]. Further discussions for how side-conditions for composite refactorings can be computed manually is shown in [19]. However, there has not been much work done on computing side-coditions for arbitrary composite refactorings automatically.

- The number of possible refactorings to implement seems endless and no tool developer will ever be able to implement refactorings for every programmer's needs. HaRe aims to provide a set of *core* refactorings, but even these are not sufficient in satisfying every possibility. Providing a user with the ability to compose their own refactorings does not achieve the same effect as providing a user with the ability to define his or her own refactorings.

  It could be possible to research into a language for defining refactorings from scratch. Rather than designing refactorings in HaRe using the API, it may be better to provide a high-level language specifically for the creation of refactorings and their pre and post conditions. The language would have to contain constructs to be able to specify a set of rules for transforming a program, and also a set of constructs for specifying the conditions of the refactoring. A compiler to convert the refactoring source files into HaRe target files would also have to be investigated, as well as the addition of an interface in HaRe to read refactoring source files.

- More interaction between HaRe and the user. Interaction between HaRe and the user during the refactoring process will allow the users to provide the refactorer with more information to allow the refactorer to be more specific when performing refactorings. Chapter 5, introduced the notion of a *for one* and a *for all* mode for refactorings; it is also possible to introduce a *for some* mode (such as in the clone detector, described in Chapter 6) which allows HaRe to cycle through a list of possibilities, allowing the user to decide whether or not to perform the refactoring in that instance or not. Wrangler, the Erlang refactoring tool also uses a step-by-step approach for clone extraction. The current refactorings for HaRe could be re-implemented using the step-by-step approach.

- Applying the research results to other functional programming languages is an important topic of future research. Currently the Kent Group is also researching into refactoring the Erlang functional language, by means of the tool Wrangler [61]. The work could just as easily be extended to other functional paradigms. The Epigram project [78] at Nottingham has opportunities for a dependently typed refactoring system. Also, it would be very interesting to see how refactoring could be used to maintain programs at a visual level —and the analogies this has with circuit design— rather than at a code level.

- In the future it is planned that dead code elimination will be expanded to take a whole program into account and not just a selected function. Dead code elimination could remove the parts of the program that are not directly related to the `main` function, which would allow the process to be extended to the whole of a large Haskell project.

- It is also planned to modify the splitter to analyze the whole scope of a program rather than simply tuple-returning functions in the scope of a particular function of interest. It is also planned to modify the splitter so that tuple-returning functions that are called from outside a function's scope can be extracted.

- Further work will detail work on refactoring with monads. It is intended for the work on slicing to be extended further to investigate backwards, dynamic slicing and forwards slicing (both static and dynamic). There are many more exciting possibilities to analyze for the refactoring of Haskell code; and it is hoped to continue work in these directions.

Appendix A

Strafunski

A.1 Traversal Scheme Excerpts

```
---------------------------------------------------------------
-- * Recursive traversal
-- * Authors: Joost Visser and Ralph Lammel.

---------------------------------------------------------------
-- ** Full traversals
--
--    * td -- top-down
--    * bu -- bottom-up

-- | Full type-preserving traversal in top-down order.
full_tdTP       :: Monad m => TP m -> TP m
full_tdTP s     =  s `seqTP` (allTP (full_tdTP s))

-- | Full type-preserving traversal in bottom-up order.
full_buTP       :: Monad m => TP m -> TP m
full_buTP s     =  (allTP (full_buTP s)) `seqTP` s

-- | Full type-unifying traversal in top-down order.
full_tdTU       :: (Monad m, Monoid a) => TU a m -> TU a m
full_tdTU s     =  op2TU mappend s (allTU' (full_tdTU s))

---------------------------------------------------------------
-- ** Traversals with stop conditions
-- | Top-down type-preserving traversal that is cut off below nodes
--   where the argument strategy succeeds.
stop_tdTP       :: MonadPlus m => TP m -> TP m
stop_tdTP s     =  s `choiceTP` (allTP (stop_tdTP s))

-- | Top-down type-unifying traversal that is cut off below nodes
--   where the argument strategy succeeds.
stop_tdTU       :: (MonadPlus m, Monoid a) => TU a m -> TU a m
stop_tdTU s     =  s `choiceTU` (allTU' (stop_tdTU s))

------------------------------------------------------------------
-- ** Single hit traversal

-- | Top-down type-preserving traversal that performs its argument
--   strategy at most once.
once_tdTP       :: MonadPlus m => TP m -> TP m
once_tdTP s     =  s `choiceTP` (oneTP (once_tdTP s))

-- | Top-down type-unifying traversal that performs its argument
--   strategy at most once.
once_tdTU       :: MonadPlus m => TU a m -> TU a m
once_tdTU s     =  s `choiceTU` (oneTU (once_tdTU s))

-- | Bottom-up type-preserving traversal that performs its argument
--   strategy at most once.
once_buTP       :: MonadPlus m => TP m -> TP m
once_buTP s     =  (oneTP (once_buTP s)) `choiceTP` s

-- | Bottom-up type-unifying traversal that performs its argument
--   strategy at most once.
once_buTU       :: MonadPlus m => TU a m -> TU a m
once_buTU s     =  (oneTU (once_buTU s)) `choiceTU` s
```

Appendix B

The Definition of PNT, from Programatica

```
-- The definition of PNT and its comprising data types.
data PNT = PNT PName (IdTy PId) OptSrcLoc

data PN i = PN i Orig

type PName = PN HsName

type PId = PN Id

data HsName = Qual ModuleName Id
            | UnQual Id

type Id = String

data Orig = L Int
          | G ModuleName Id OptSrcLoc
          | D Int OptSrcLoc
          | S SrcLoc
          | Sn Id SrcLoc
          | P

data ModuleName = PlainModule String

data ModuleName = PlainModule String
                   | MainModule FilePath

newtype OptSrcLoc = N (Maybe SrcLoc)

data SrcLoc = SrcLoc {srcPath :: FilePath
               srcChar, srcLine, srcColumn :: !Int}
```

```
data IdTy i = Value
              | FieldOf i (TypeInfo i)
              | MethodOf i [i]
              | ConstrOf i (TypeInfo i)
              | Class [i]
              | Type (TypeInfo i)
              | Assertion
              | Property

data TypeInfo i = TypeInfo { defType :: (Maybe DefTy)
                  constructors :: [ConInfo i]
                  fields :: [i]
                  }

data DefTy = Newtype
              | Data
              | Synonym
              | Primitive

data ConInfo i = ConInfo { conName :: i
                  conArity :: Int
                  conFields :: (Maybe [i])
                  }
```

Appendix C

Huffman Tree Encoding

```
--         Main.lhs

--         The main module of the Huffman example

--         (c) Simon Thompson, 1995,1998.


-- The main module of the Huffman example

module Main (main) where

import Types    ( Tree(Leaf,Node), Bit(L,R), HCode , Table )
import Coding   ( codeMessage, decodeMessage )
import MakeCode ( codes, codeTable )

main = print decoded


-- Examples
-- ^^^^^^^^

-- The coding table generated from the text "there is a green hill".

tableEx :: Table
tableEx = codeTable (codes "there is a green hill")

-- The Huffman tree generated from the text "there is a green hill",
-- from which tableEx is produced by applying codeTable.

treeEx :: Tree
treeEx = codes "there is a green hill"
```

```
-- A message to be coded.

message :: String
message = "there are green hills here"

-- The message in code.

coded :: HCode
coded = codeMessage tableEx message

-- The coded message decoded.

decoded :: String
decoded = decodeMessage treeEx coded
```

```
--          Codetable.lhs
--
--          Converting a Huffman tree to a ord table.
--
--          (c) Simon Thompson, 1995, 1998.
--

module CodeTable ( codeTable ) where
import Types ( Tree(Leaf,Node), Bit(L,R), HCode, Table )

-- Making a table from a Huffman tree.

codeTable :: Tree -> Table

codeTable = convert []

-- Auxiliary function used in conversion to a table. The first argument is
-- the HCode which codes the path in the tree to the current Node, and so
-- codeTable is initialised with an empty such sequence.

convert :: HCode -> Tree -> Table

convert cd (Leaf c n) =  [(c,cd)]
convert cd (Node n t1 t2)
 = (convert (cd++[L]) t1) ++ (convert (cd++[R]) t2)

-- Show functions
-- ^^^^^^^^^^^^^^

-- Show a tree, using indentation to show structure.
--
showTree :: Tree -> String

showTree t = showTreeIndent 0 t

-- The auxiliary function showTreeIndent has a second, current
-- level of indentation, as a parameter.

showTreeIndent :: Int -> Tree -> String
showTreeIndent m (Leaf c n)
  = spaces m ++ show c ++ " " ++ show n ++ "\n"
showTreeIndent m (Node n t1 t2)
  = showTreeIndent (m+4) t1 ++
    spaces m ++ "[" ++ show n ++ "]" ++ "\n" ++
    showTreeIndent (m+4) t2

-- A String of n spaces.

spaces :: Int -> String

spaces n = replicate n ' '

-- To show a sequence of Bits.

showCode :: HCode -> String
showCode = map conv
           where
           conv R = 'R'
           conv L = 'L'

-- To show a table of codes.

showTable :: Table -> String
showTable
  = concat . map showPair
    where
    showPair (ch,co) = [ch] ++ " " ++ showCode co ++ "\n"
```

```
--
--         Coding.lhs
--
--         Huffman coding in Haskell.
--         The top-level functions for coding and decoding.
--
--         (c) Simon Thompson, 1995.


module Coding ( codeMessage , decodeMessage ) where

import Types ( Tree(Leaf,Node), Bit(L,R), HCode, Table )

-- Code a message according to a table of codes.

codeMessage :: Table -> [Char] -> HCode

codeMessage tbl = concat . map (lookupTable tbl)

-- lookupTable looks up the meaning of an individual char in
-- a Table.

lookupTable :: Table -> Char -> HCode

lookupTable [] c = error "lookupTable"
lookupTable ((ch,n):tb) c
  | (ch==c)     = n
  | otherwise   = lookupTable tb c


-- Decode a message according to a tree.
--
-- The first tree arguent is constant, being the tree of codes;
-- the second represents the current position in the tree relative
-- to the (partial) HCode read so far.


decodeMessage :: Tree -> HCode -> String

decodeMessage tr
  = decodeByt tr
    where

    decodeByt (Node n t1 t2) (L:rest)
        = decodeByt t1 rest
```

```
decodeByt (Node n t1 t2) (R:rest)
    = decodeByt t2 rest

decodeByt (Leaf c n) rest
    = c : decodeByt tr rest

decodeByt t [] = []
```

```
--
--          Frequency.lhs
--
--          Calculating the frequencies of words in a text, used in
--          Huffman coding.
--
--          (c) Simon Thompson, 1995, 1998.
--

module Frequency ( frequency ) where
import CodeTable2
-- Calculate the frequencies of characters in a list.
--
-- This is done by sorting, then counting the number of
-- repetitions. The counting is made part of the merge
-- operation in a merge sort.

frequency :: [Char] -> [ (Char,Int) ]

frequency
  = mergeSort freqMerge . mergeSort alphaMerge . map start
    where
    start ch = (ch,1)

-- Merge sort parametrised on the merge operation. This is more
-- general than parametrising on the ordering operation, since
-- it permits amalgamation of elements with equal keys
-- for instance.
--
mergeSort :: ([a]->[a]->[a]) -> [a] -> [a]

mergeSort merge xs
  | length xs < 2         = xs
  | otherwise
      = merge (mergeSort merge first)
              (mergeSort merge second)
        where
        first  = take half xs
        second = drop half xs
        half   = (length xs) `div` 2

-- Order on first entry of pairs, with
-- accumulation of the numeric entries when equal first entry.

alphaMerge :: [(Char,Int)] -> [(Char,Int)] -> [(Char,Int)]
alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)
  | (p==q)         = (p,n+m) : alphaMerge xs ys
  | (p<q)         = (p,n) : alphaMerge xs ((q,m):ys)
  | otherwise         = (q,m) : alphaMerge ((p,n):xs) ys

-- Lexicographic ordering, second field more significant.
--
freqMerge :: [(Char,Int)] -> [(Char,Int)] -> [(Char,Int)]

freqMerge xs [] = xs
freqMerge [] ys = ys
freqMerge ((p,n):xs) ((q,m):ys)
  | (n<m || (n==m && p<q))
    = (p,n) : freqMerge xs ((q,m):ys)
  | otherwise
    = (q,m) : freqMerge ((p,n):xs) ys
```

```
--
--          MakeCode.lhs
--
--          Huffman coding in Haskell.
--
--          (c) Simon Thompson, 1995, 1998.
--

module MakeCode ( codes, codeTable ) where

import Types
import Frequency ( frequency )
import MakeTree  ( makeTree )
import CodeTable ( codeTable )

-- Putting together frequency calculation and tree conversion

codes :: [Char] -> Tree

codes = makeTree . frequency
```

```
--
--         makeTree.lhs
--
--         Turn a frequency table into a Huffman tree
--
--         (c) Simon Thompson, 1995.
--

module MakeTree ( makeTree ) where

import Types ( Tree(Leaf,Node), Bit(L,R), HCode, Table )

-- Convert the trees to a list, then amalgamate into a single
-- tree.

makeTree :: [ (Char,Int) ] -> Tree

makeTree = makeCodes . toTreeList

-- Huffman codes are created bottom up: look for the least
-- two frequent letters, make these a new "isAlpha" (i.e. tree)
-- and repeat until one tree formed.

-- The function toTreeList makes the initial data structure.

toTreeList :: [ (Char,Int) ] -> [ Tree ]

toTreeList = map (uncurry Leaf)

-- The value of a tree.

value :: Tree -> Int

value (Leaf _ n)   = n
value (Node n _ _) = n

-- Pair two trees.

pair :: Tree -> Tree -> Tree

pair t1 t2 = Node (v1+v2) t1 t2
             where
             v1 = value t1
             v2 = value t2
```

```
-- Insert a tree in a list of trees sorted by ascending value.

insTree :: Tree -> [Tree] -> [Tree]

insTree t [] = [t]
insTree t (t1:ts)
  | (value t <= value t1)    = t:t1:ts
  | otherwise                = t1 : insTree t ts
--
-- Amalgamate the front two elements of the list of trees.

amalgamate :: [ Tree ] -> [ Tree ]

amalgamate ( t1 : t2 : ts )
  = insTree (pair t1 t2) ts

-- Make codes: amalgamate the whole list.

makeCodes :: [Tree] -> Tree

makeCodes [t] = t
makeCodes ts = makeCodes (amalgamate ts)
```

```
--         Types.lhs
--
--         The types used in the Huffman coding example.
--
--         (c) Simon Thompson, 1995, 1998


-- The interface to the module Types is written out
-- explicitly here, after the module name.

module Types ( Tree(Leaf,Node), Bit(L,R),
               HCode , Table  ) where

-- Trees to represent the relative frequencies of characters
-- and therefore the Huffman codes.

data Tree = Leaf Char Int | Node Int Tree Tree

-- The types of bits, Huffman codes and tables of Huffman codes.

data Bit = L | R deriving (Eq,Show)

type HCode = [Bit]

type Table = [ (Char,HCode) ]
```

Appendix D

Clone Report for Huffman Coding

```
/home/cmb21/huffman/CodeTable.hs  ((70,5),(70,18)): >concat . map showPair<
/home/cmb21/huffman/Coding.hs  ((18,19),(18,48)): >concat . map (lookupTable tbl)<
2 occurrences.


/home/cmb21/huffman/Frequency.hs  ((61,22),(61,24)): >p<q<
/home/cmb21/huffman/Frequency.hs  ((61,6),(61,8)): >n<m<
2 occurrences.


/home/cmb21/huffman/CodeTable.hs  ((46,5),(46,44)):
            >spaces m ++ show c ++ " " ++ show n ++ "\n"<
/home/cmb21/huffman/CodeTable.hs  ((49,5),(50,26)):
          >spaces m ++ "[" ++ show n ++ "]" ++ "\n" ++
    showTreeIndent (m+4) t2<
2 occurrences.


/home/cmb21/huffman/Frequency.hs  ((51,19),(51,50)): >(p,n) : alphaMerge xs ((q,m):ys)<
/home/cmb21/huffman/Frequency.hs  ((50,19),(50,43)): >(p,n+m) : alphaMerge xs ys<
/home/cmb21/huffman/Frequency.hs  ((52,19),(52,49)): >(q,m) : alphaMerge ((p,n):xs) ys<
3 occurrences.


/home/cmb21/huffman/Frequency.hs  ((62,7),(62,37)): >(p,n) : freqMerge xs ((q,m):ys)<
/home/cmb21/huffman/Frequency.hs  ((64,7),(64,36)): >(q,m) : freqMerge ((p,n):xs) ys<
2 occurrences.


/home/cmb21/huffman/CodeTable.hs  ((39,14),(39,31)): >showTreeIndent 0 t<
/home/cmb21/huffman/CodeTable.hs  ((48,5),(48,26)): >showTreeIndent (m+4) t1<
2 occurrences.


/home/cmb21/huffman/MakeTree.hs  ((53,37),(53,47)): >insTree t ts<
/home/cmb21/huffman/MakeTree.hs  ((60,5),(60,26)): >insTree (pair t1 t2) ts<
```

```
2 occurrences.


/home/cmb21/huffman/Coding.hs  ((51,15),(51,28)): >decodeByt tr rest<
/home/cmb21/huffman/Coding.hs  ((45,11),(45,24)): >decodeByt t1 rest<
/home/cmb21/huffman/Coding.hs  ((48,11),(48,24)): >decodeByt t2 rest<
/home/cmb21/huffman/Frequency.hs  ((35,9),(36,38)): >merge (mergeSort merge first)
              (mergeSort merge second)<
4 occurrences.


/home/cmb21/huffman/CodeTable.hs  ((25,27),(25,32)): >(c,cd)<
/home/cmb21/huffman/Frequency.hs  ((23,16),(23,21)): >(ch,1)<
2 occurrences.


/home/cmb21/huffman/Coding.hs  ((27,5),(27,11)): >(ch==c)<
/home/cmb21/huffman/Frequency.hs  ((50,5),(50,10)): >(p==q)<
2 occurrences.


/home/cmb21/huffman/CodeTable.hs  ((27,4),(27,25)): >(convert (cd++[L]) t1)<
/home/cmb21/huffman/CodeTable.hs  ((27,30),(27,51)): >(convert (cd++[R]) t2)<
2 occurrences.
```

Appendix E

Clone Report for `MainNew` within `nhc`

```
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((554,30),(554,40)):
>foreigns++newforeigns<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((287,24),(287,73)):
>"Actual imports used by this module ("++modname++"):"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((294,40),(296,59)):
 >"Couldn't write interface file "
                                      ++ sTypeFile flags ++ ":"
                                      ++ show ioerror ++ "\n"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((587,23),(587,58)):
>"Intermediate need after import "++fname<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((589,24),(589,67)):
>"Intermediate symbol table after import "++fname<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((591,25),(591,68)):
>"Intermediate rename table after import "++fname<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((599,20),(599,50)):
>strIS state v ++ "{"++show v++"}"<
7 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((578,15),(578,72)):
>mixLine (map show (listAT (getRenameTableIS importState)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((592,9),(592,66)):
>mixLine (map show (listAT (getRenameTableIS importState)))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((576,15),(576,72)):
>mixLine (map show (listAT (getSymbolTableIS importState)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((590,9),(590,66)):
>mixLine (map show (listAT (getSymbolTableIS importState)))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((191,10),(191,59)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((218,11),(218,60)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((232,13),(232,62)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((280,11),(280,60)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((312,12),(312,61)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((333,12),(333,61)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((344,12),(344,61)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((372,7),(372,56)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((390,7),(390,56)):
 >mixLine (map show (listAT (getSymbolTable state)))<
9 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((412,35),(412,56)):
>return (eslabs,escode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((428,13),(428,35)):
 >return (eslabs, escode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((553,19),(553,40)):
>return (eslabs,escode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((413,21),(417,28)):
>return (foldr (\a b-> gcodeGather Labels state b a)
                                (emitWord Labels "42" eslabs) zcons
                         ,foldr (\a b-> gcodeGather Code   state b a)
                                (emitWord Code "42" escode) zcons
                         )<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((543,17),(544,62)):
>return (gcodeGather Labels state eslabs gcode
                     ,gcodeGather Code   state escode gcode)<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((359,14),(359,42)):
>return (stgArity state decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((378,14),(378,42)):
>return (stgArity state decls)<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((574,15),(574,57)):
>show (listM (thd3 (getNeedIS importState)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((588,9),(588,51)):
>show (listM (thd3 (getNeedIS importState)))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((98,25),(98,33)):
>pF True errmsg<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((328,7),(328,15)):
>pF True "Warning pattern removal"<
```

```
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((262,3),(262,19)):
>pF (sScc flags) "Declarations after scc:"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((264,3),(264,19)):
>pF (sScc flags) "Class/instances after scc:"<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((475,12),(475,27)):
>hPutStr handle "\n#include <haskell2c.h>\n#include <HsFFI.h>\n"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((419,20),(421,57)):
>hPutStr handle (gcodeHeader
                     (foldr ( \ a b -> foldr (gcodeDump state) b a)
                                                "\n" zcons))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((476,24),(476,55)):
>hPutStr handle (strForeign f "")<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((464,17),(465,49)):
>hPutStr handle (foldr (\a b -> foldr (gcodeDump state) b a)
                                         "\n" gcode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((546,26),(546,76)):
>hPutStr handle (foldr (gcodeDump state) "\n" gcode)<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((458,18),(458,59)):
>mapM_ (hPutStr handle) (emit Code escode')<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((457,18),(457,61)):
>mapM_ (hPutStr handle) (emit Labels eslabs')<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((514,31),(514,58)):
>concatMap (strGcode state) gcode<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((520,42),(520,69)):
>concatMap (strGcode state) gcode<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((525,42),(525,69)):
>concatMap (strGcode state) gcode<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((530,40),(530,67)):
>concatMap (strGcode state) gcode<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((535,42),(535,69)):
 >concatMap (strGcode state) gcode<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((331,12),(331,38)):
>strPCode (strISInt state) decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((342,12),(342,38)):
>strPCode (strISInt state) decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((353,7),(353,33)):
>strPCode (strISInt state) decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((370,7),(370,33)):
>strPCode (strISInt state) decls<
```

```
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((388,48),(388,74)):
 >strPCode (strISInt state) decls<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((157,16),(157,63)):
>catchError info ("In file: "++sSourceFile flags)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((579,3),(579,37)):
 >catchError (getErrIS importState) "Errors after importing module"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((131,16),(132,58)):
>catchError (parseit parseProg lexdata)
                         ("In file: "++sSourceFile flags)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((213,15),(214,26)):
>catchError (derive tidFun state derived decls)
                        "Deriving failed"<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((185,11),(187,22)):
>catchError (rename flags modid qualFun expFun inf decls
                            importState overlap)
                    "Errors when renaming"<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((442,15),(442,35)):
>generateCode handle flags<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((554,3),(554,23)):
>generateCode handle flags<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((360,3),(361,38)):
>pF (sArity flags) "Declarations after first arity grouping"
     (strPCode (strISInt state) decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((379,3),(380,38)):
>pF (sArity flags) "Declarations after second arity grouping"
     (strPCode (strISInt state) decls)<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((446,3),(447,51)):
>pF (sGcodeRel flags) "G Code (rel)"
     (concatMap (strGcodeRel state) (concat gcode))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((539,3),(539,75)):
>pF (sGcodeRel flags) "G Code (rel)" (concatMap (strGcodeRel state) gcode)<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((192,3),(192,58)):
>catchError (getErrorsIS state) "Errors after renaming" mixLine<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((233,3),(233,63)):
>catchError (getErrorsIS state) "Errors after extract phase" mixLine<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((247,3),(247,65)):
>catchError (getErrorsIS state) "Errors after removing fields" mixLine<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((281,3),(281,73)):
>catchError (getErrorsIS state) "Errors after type inference/checking" mixLine<
```

```
4 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((165,18),(166,28)):
>nhcImport flags (addPreludeTupleInstances () (initIS need))
                           imports<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((593,3),(593,31)):
 >nhcImport flags importState xs<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((455,36),(455,63)):
>gcodeGather Code   state b a<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((454,36),(454,63)):
>gcodeGather Labels state b a<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((189,10),(189,47)):
>prettyPrintId flags state ppTopDecls decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((216,12),(216,49)):
 >prettyPrintId flags state ppTopDecls decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((246,11),(246,48)):
>prettyPrintId flags state ppTopDecls decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((263,6),(263,43)):
>prettyPrintId flags state ppTopDecls decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((278,11),(278,48)):
>prettyPrintId flags state ppTopDecls decls<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((310,12),(310,66)):
 >prettyPrintId flags state ppTopDecls (DeclsParse decls)<
6 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((402,34),(405,34)):
>hPutStr stderr ("Couldn't open object file "
                                   ++ sObjectFile flags ++ ":"
                                   ++ show ioerror ++ "\n")
                                 exit<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((423,34),(427,34)):
 >hPutStr stderr
                                    ("Failed writing to object file "
                                    ++ sObjectFile flags ++ ":"
                                    ++ show ioerror ++ "\n")
                                 exit<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((459,30),(463,30)):
>hPutStr stderr
                                 ("Failed writing code to object file "
                                  ++ sObjectFile flags ++ ":"
                                  ++ show ioerror ++ "\n")
                             exit<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((467,31),(471,31)):
>hPutStr stderr
                                ("Failed appending tables to object file "
                                 ++ sObjectFile flags ++ ":" ++
                                show ioerror ++ "\n")
                             exit<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((548,40),(552,40)):
>hPutStr stderr
                                        ("Failed appending to object file "
                                         ++ sObjectFile flags ++ ":"
                                         ++ show ioerror ++ "\n")
                                      exit<
5 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((328,41),(328,56)):
>(mixLine errors)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((288,10),(288,50)):
 >(mixLine (reportFnImports modname state))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((98,40),(98,56)):
>(showErrors errs)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((230,22),(230,42)):
>(extract decls state)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((445,19),(445,49)):
>(gcodeFixFinish state fixState)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((435,24),(435,49)):
>(gcodeFixInit state flags)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((524,21),(524,43)):
>(gcodeOpt1 state gcode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((534,21),(534,43)):
>(gcodeOpt2 state gcode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((149,20),(149,45)):
>(needProg flags parsedPrg)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((387,21),(387,41)):
>(posAtom state decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((292,9),(292,72)):
>(writeFile (sTypeFile flags) (buildInterface flags modid state))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((125,18),(125,70)):
>(map (\ (p,l,_,_) -> strPos p ++ ':':show l) lexdata)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((206,24),(206,52)):
 >(ffiTrans decls tidFun state)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((308,19),(308,48)):
>(fixSyntax decls state tidFun)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((351,21),(351,59)):
>(freeVar (sKeepCase flags) decls state)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((529,21),(529,59)):
>(gcodeMem (sProfile flags) state gcode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((399,19),(399,57)):
>(gcodeZCon (sProfile flags) state zcon)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((120,22),(123,47)):
>(lexical (sUnderscore flags) (sSourceFile flags)
                             (if sUnlit flags
                               then unlit (sSourceFile flags) mainChar
                               else mainChar))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((368,21),(368,49)):
```

```
>(liftCode decls state tidFun)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((133,29),(133,73)):
 >(prettyPrintTokenId flags ppModule parsedPrg)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((244,21),(244,52)):
>(removeDecls decls tidFun state)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((259,19),(259,48)):
>(rmClasses tidFun state decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((513,21),(513,58)):
>(stgGcode (sProfile flags) state decl)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((519,21),(519,57)):
>(gcodeFix flags state fixState gcode)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((265,5),(265,49)):
>(prettyPrintId flags state ppClassCodes code)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((340,21),(340,64)):
 >(primCode primFlags True tidFun state decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((276,21),(276,70)):
>(typeTopDecls tidFun userDefault state code decls)<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((322,21),(325,63)):
>(caseTopLevel (if sPrelude flags
                                   then "Prelude:"++ sSourceFile flags
                                   else reverse (unpackPS (mrpsIS state)))
                                   t2i code decls state tidFun)<
26 occurrences.
```

Appendix F

Clone Class Extract

```
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((578,15),(578,72)):
 >mixLine (map show (listAT (getRenameTableIS importState)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((592,9),(592,66)):
>mixLine (map show (listAT (getRenameTableIS importState)))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((576,15),(576,72)):
>mixLine (map show (listAT (getSymbolTableIS importState)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((590,9),(590,66)):
 >mixLine (map show (listAT (getSymbolTableIS importState)))<
2 occurrences.


/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((191,10),(191,59)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((218,11),(218,60)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((232,13),(232,62)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((280,11),(280,60)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((312,12),(312,61)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((333,12),(333,61)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((344,12),(344,61)):
 >mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((372,7),(372,56)):
>mixLine (map show (listAT (getSymbolTable state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((390,7),(390,56)):
>mixLine (map show (listAT (getSymbolTable state)))<
9 occurrences.
```

Appendix G

Folding As Patterns Implementation

```
-----------------------------------------------------------------------------
-- |
-- Module      :  RefacAsPatterns
-- Copyright   :  (c) Christopher Brown 2007
--
-- Maintainer  :  cmb21@kent.ac.uk
-- Stability   :  provisional
-- Portability :  portable
--
-- This module contains a transformation for HaRe.
-- Convert all patterns into AS patterns and
-- change all reference to the pattern on the RHS to references
-- to the AS name.
--
-----------------------------------------------------------------------------

module RefacAsPatterns where


import System.IO.Unsafe
import PrettyPrint
import RefacTypeSyn
import RefacLocUtils
import GHC (Session)
import Data.Char
import GHC.Unicode
import AbstractIO
import Maybe
import List
import RefacUtils

data Patt = Match HsMatchP | MyAlt HsAltP | MyDo HsStmtP | MyListComp HsExpP
             deriving (Show)
refacAsPatterns args
  = do let
           fileName    = args!!0
           name        = args!!1
           beginRow    = read (args!!2)::Int
           beginCol    = read (args!!3)::Int
           endRow      = read (args!!4)::Int
           endCol      = read (args!!5)::Int

       AbstractIO.putStrLn "refacAsPatterns"

       unless (isVarId name)
                $ error "The new name is invalid!\n"
       -- Parse the input file.
       modInfo@(inscps, exps, mod, tokList) <- parseSourceFile fileName

       -- there are two modes to this refactoring:
       -- either one selects a pattern, or, one selects an expression (referring to a pattern).

       let exp = locToExp (beginRow, beginCol) (endRow, endCol) tokList mod

       case exp of
         Exp (HsId (HsVar (PNT (PN (UnQual "unknown") (G (PlainModule "unknown")
         "--" (N Nothing))) Value  (N Nothing)) ))
           -> do

                let (pnt, pat, match) = findPattern tokList (beginRow, beginCol) (endRow, endCol) mod

                ((_,m), (newToks, newMod)) <- applyRefac (changePattern name pat match)
                (Just (inscps, exps, mod, tokList)) fileName

                unless (newMod /= mod) $ AbstractIO.putStrLn "Pattern does not occur on the rhs!"

                writeRefactoredFiles False [((fileName, m), (newToks, newMod))]
                AbstractIO.putStrLn "Completed.\n"

         _
           -> do
                 -- all we need is the expression and the pattern to which it is imitating...
                 let pat = getPat exp mod

                 ((_,m), (newToks, newMod)) <- applyRefac (changePatternSimple name pat exp)
                  (Just (inscps, exps, mod, tokList)) fileName

                 writeRefactoredFiles False [((fileName, m), (newToks, newMod))]
                 AbstractIO.putStrLn "Completed.\n"


getPat exp t
 = fromMaybe (error "No Pattern is associated with the highlighted expression!")
     (applyTU (once_tdTU (failTU `adhocTU` worker)) t)
       where
        worker (p :: HsPatP)
          | rewritePats p == exp = Just p
          | otherwise = Nothing


convertPat :: String -> HsPatP -> HsPatP
convertPat _ (Pat (HsPAsPat n p)) = error "The selected pattern is already an as-pattern!"
convertPat _ (Pat (HsPId _)) = error "Cannot perform to-as-patterns on a simple variable!"
convertPat _ (Pat (HsPLit _ _)) = error "Cannot perform to-as-patterns on a constant value!"
convertPat name (Pat (HsPParen p)) = convertPat name p
convertPat name p = (Pat (HsPAsPat (nameToPNT name) p))

changePatternSimple name pat exp (_, _, t)
 = do

       inscopeNames <- hsVisibleNames exp t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       -- let newPats = checkPats name pat [p]
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat exp

       newT <- update exp newExp t
       newT2 <- update pat convertedPat newT

       return newT2
```

```
changePattern name pat (Match match) (_, _,t)
  = do
       newDecl <- lookInMatches t name pat [match]
       newT <- update match (newDecl !! 0) t
       return newT

changePattern name pat (MyAlt alt) (_, _, t)
  = do
       newAlt <- lookInAlt t name pat alt
       newT <- update alt newAlt t
       return newT

changePattern name pat (MyDo inDo) (_,_, t)
  = do
       newDo <- lookInDo t name pat inDo
       newT <- update inDo newDo t
       return newT

changePattern name pat (MyListComp inList) (_,_,t)
 = do
       newList <- lookInList t name pat inList
       newT <- update inList newList t
       return newT

convertPatterns t name pat dec@(Dec (HsPatBind a b (HsBody e) ds))
  = do
        newExp <- checkExpr (Pat (HsPAsPat (nameToPNT name) pat)) e
        return (Dec (HsPatBind a b (HsBody newExp) ds))
convertPatterns t name pat (Dec (HsFunBind loc matches))
  =
    do
       rest <- lookInMatches t name pat matches
       return (Dec (HsFunBind loc  rest ))

lookInMatches _ _ _ []  = return []
lookInMatches t name pat (match@(HsMatch s pnt (p:ps) (HsGuard g@((_,_,e):gs)) ds):ms)
    = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat (p:ps)
       let convertedPat = convertPat name pat
       newGuard <- checkGuards convertedPat g
       newDS <- mapM (convertPatterns t name pat) ds
       rest <- lookInMatches t name pat ms
       if newGuard == g && newDS == ds
         then do
           return (HsMatch s pnt (p:ps) (HsGuard g) ds : rest)
         else do
           return (HsMatch s pnt newPats (HsGuard newGuard) newDS : rest)


lookInMatches t name pat (match@(HsMatch s pnt (p:ps) (HsBody e) ds):ms)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat (p:ps)
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- mapM (convertPatterns t name pat) ds
       rest <- lookInMatches t name pat ms
       if newExp == e && newDS == ds
         then do
           return (HsMatch s pnt (p:ps) (HsBody e) ds : rest)
         else do
```

```
          return (HsMatch s pnt newPats (HsBody newExp) newDS : rest)


lookInAlt t name pat (alt@(HsAlt s p (HsGuard g@((_,_,e):gs))  ds)::HsAltP)
    = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat [p]
       let convertedPat = convertPat name pat
       newGuard <- checkGuards convertedPat g
       newDS <- mapM (convertPatterns t name pat) ds
       -- rest <- lookInMatches t name pat ms
       if newGuard == g && newDS == ds
         then do
           return (HsAlt s p (HsGuard g) ds)
         else do
           return (HsAlt s (ghead "lookInAlt" newPats) (HsGuard newGuard) newDS)

lookInAlt t name pat (alt@(HsAlt s p (HsBody e)  ds)::HsAltP)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat [p]
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- mapM (convertPatterns t name pat) ds
       -- rest <- lookInMatches t name pat ms
       if newExp == e && newDS == ds
         then do
           return (HsAlt s p (HsBody e) ds)
         else do
           return (HsAlt s (ghead "lookInAlt" newPats) (HsBody newExp) newDS)

lookInDo t name pat (inDo@(HsGenerator s p e rest))
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat [p]
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- lookInExps t name pat rest
       -- rest <- lookInMatches t name pat ms
       if newExp == e && newDS == rest
         then do
           return (HsGenerator s p e rest)
         else do
           return (HsGenerator s (ghead "lookInDo" newPats) newExp newDS)

lookInList t name pat (inList@(Exp (HsListComp d)))
 = do
      res <- lookInDo t name pat d
      return (Exp (HsListComp res))

lookInExps t name pat (HsLast e)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       return (HsLast newExp)
```

```
lookInExps t name pat (HsGenerator s p e rest)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")

       let newPats = checkPats name pat [p]
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- lookInExps t name pat rest
       -- rest <- lookInMatches t name pat ms
       if newExp == e && newDS == rest
         then do
           return (HsGenerator s p e rest)
         else do
           return (HsGenerator s (ghead "lookInDo" newPats) newExp newDS)

lookInExps t name pat (HsQualifier e rest)
   = do
       inscopeNames <- hsVisibleNames e t
       unless (not (name `elem` inscopeNames))
        $ error ("the use of the name: " ++ name ++ " is already in scope!")
       let convertedPat = convertPat name pat
       newExp <- checkExpr convertedPat e
       newDS <- lookInExps t name pat rest
       return (HsQualifier newExp newDS)

lookInExps t name pat (HsLetStmt ds rest)
   = do

       let convertedPat = convertPat name pat
       newRest <- lookInExps t name pat rest
       newDS <- mapM (convertPatterns t name pat) ds

       -- rest <- lookInMatches t name pat ms
       if newDS == ds && rest == newRest
         then do
           return (HsLetStmt ds rest)
         else do
           return (HsLetStmt newDS newRest)

checkPats :: String -> HsPatP -> [ HsPatP ] -> [ HsPatP ]
checkPats _ _ [] = []
checkPats name pat (pat2@(Pat (HsPParen p)): ps)
 | pat == pat2  = (Pat (HsPParen (Pat (HsPAsPat (nameToPNT name) p)))) : checkPats name pat ps
 | otherwise =  (Pat (HsPParen res)) : (checkPats name pat ps)
     where
       res = ghead "checkPats HsPParen res" $ checkPats name pat [p]
checkPats name pat (pat2@(Pat (HsPApp i ps)):pss)
 | pat == pat2 = (Pat (HsPAsPat (nameToPNT name) pat2)) : checkPats name pat pss
 | otherwise = (Pat (HsPApp i (checkPats name pat ps))) : checkPats name pat pss

checkPats name pat (pat2@(Pat (HsPTuple s ps)):pss)
 | pat == pat2 = (Pat (HsPAsPat (nameToPNT name) pat2)) : checkPats name pat pss
 | otherwise = (Pat (HsPTuple s (checkPats name pat ps))) : checkPats name pat pss

checkPats name pat (pat2@(Pat (HsPList s ps)) : pss)
 | pat == pat2 = (Pat (HsPAsPat (nameToPNT name) pat2)) : checkPats name pat pss
 | otherwise   = (Pat (HsPList s (checkPats name pat ps))) : checkPats name pat pss

checkPats name pat ((Pat (HsPInfixApp p1 i p2)) : pss)
 = (Pat (HsPInfixApp res1 i res2)) : checkPats name pat pss
     where
       res1 = ghead "checkPats HsPInFixApp res1" $ checkPats name pat [p1]
       res2 = ghead "checkPats HsPInfixApp res2" $ checkPats name pat [p2]
checkPats name pat (p:ps)
 | pat == p  = (Pat (HsPAsPat (nameToPNT name) p)) : checkPats name pat ps
 | otherwise = p : (checkPats name pat ps)
```

```
-- checkGuards [] g = return g
checkGuards _ [] = return []
checkGuards pat@(Pat (HsPAsPat s p)) ((s1, e1, e2):gs)
  = do
       -- rewrite the guard
       newGuard  <- rewriteExp s (rewritePats p) e1 pat
       -- newGuard' <- checkExpr ps newGuard

       -- rewrite the RHS of the guard
       rhs       <- rewriteExp s (rewritePats p) e2 pat
       -- rhs'      <- checkExpr ps rhs
       rest <- checkGuards pat gs

       return ((s1, newGuard, rhs):rest)

-- checkExpr :: [ HsPatP ] -> HsExpP -> m HsExpP
-- checkExpr [] e = return e
checkExpr pat@(Pat (HsPAsPat s p)) e
    = do
        newExp <- rewriteExp s (rewritePats p) e pat  --   error $ show (rewritePats p)
        -- newExp' <- checkExpr ps newExp
        return newExp
checkExpr _ e = return e

-- rewrite exp checks to see whether the given expression occurs within
-- the second exp. If it does, the expression is replaced with the name of the 'as'
-- pattern.
-- rewriteExp :: String -> HsExpP -> HsExpP -> HsExpP
rewriteExp name e1 e2 t
  = applyTP (full_tdTP (idTP `adhocTP` (inExp e1))) e2
     where
       -- inExp :: HsExpP -> HsExpP -> HsExpP
       inExp (Exp (HsParen e1)::HsExpP) (e2::HsExpP)
        = do
             -- error $ show e2
             allDef <- allDefined e2 t
             if (rmLocs e1) == (rmLocs e2) && (and allDef)
               then do
                  return (nameToExp (pNTtoName name))
               else do
                  return e2
       inExp (e1::HsExpP) (e2::HsExpP)
         | (rmLocs e1) == (rmLocs e2)  = return (nameToExp (pNTtoName name))
         | otherwise = return e2

allDefined e t
  = applyTU (full_tdTU (constTU [] `adhocTU` inPNT)) e
      where
        inPNT (p@(PNT pname ty _)::PNT)
         = return [findPN pname t]

rewritePats :: HsPatP -> HsExpP
rewritePats (pat1@(Pat (HsPRec x y)))
  = Exp (HsRecConstr loc0 x (map sortField y))
    where
      sortField :: HsFieldI a HsPatP -> HsFieldI a HsExpP
      sortField (HsField i e) = (HsField i (rewritePats e))

rewritePats (pat1@(Pat (HsPLit x y)))
  = Exp (HsLit x y)

rewritePats (pat1@(Pat (HsPAsPat i p1)))
  = Exp (HsAsPat i (rewritePats p1))

rewritePats (pat1@(Pat (HsPIrrPat p1)))
  = Exp (HsIrrPat (rewritePats p1))
```

```
rewritePats (pat1@(Pat (HsPWildCard)))
  = nameToExp "undefined"

rewritePats (pat1@(Pat (HsPApp i p1)))
  = createFuncFromPat i (map rewritePats p1)

rewritePats (pat1@(Pat (HsPList _ p1)))
  = Exp (HsList (map rewritePats p1))

rewritePats (pat1@(Pat (HsPTuple _ p1)))
  = Exp (HsTuple (map rewritePats p1))

rewritePats (pat1@(Pat (HsPInfixApp p1 x p2)))
  =  Exp (HsInfixApp (rewritePats p1)
                     (HsCon x)
                     (rewritePats p2))

rewritePats (pat@(Pat (HsPParen p1)))
  = Exp (HsParen (rewritePats p1))

rewritePats (pat1@(Pat (HsPId (HsVar (PNT (PN (UnQual (i:is)) a) b c)))))
 | isUpper i = (Exp (HsId (HsCon (PNT (PN (UnQual (i:is)) a) b c))))
 | otherwise = (Exp (HsId (HsVar (PNT (PN (UnQual (i:is)) a) b c))))
rewritePats (pat1@(Pat (HsPId (HsCon (PNT (PN (UnQual (i:is)) a) b c)))))
  = (Exp (HsId (HsCon (PNT (PN (UnQual (i:is)) a) b c))))
-- rewritePats p = error $ show p


-- strip removes whitespace and '\n' from a given string
strip :: String -> String
strip [] = []
strip (' ':xs) = strip xs
strip ('\n':xs) = strip xs
strip (x:xs) = x : strip xs

isInfixOf needle haystack = any (isPrefixOf needle) (tails haystack)

--check whether the cursor points to the beginning of the datatype declaration
--taken from RefacADT.hs
checkCursor :: String -> Int -> Int -> HsModuleP -> Either String HsDeclP
checkCursor fileName row col mod
 = case locToTypeDecl of
     Nothing -> Left ("Invalid cursor position. Please place cursor at the beginning of the definition!")
     Just decl -> Right decl
   where
    locToTypeDecl = find (definesPNT (locToPNT fileName (row, col) mod)) (hsModDecls mod)

    definesPNT pnt d@(Dec (HsPatBind loc p e ds))
      = findPNT pnt d
    definesPNT pnt d@(Dec (HsFunBind loc ms)) = findPNT pnt d
    definesPNT pnt _ = False

{-|
Takes the position of the highlighted code and returns
the function name, the list of arguments, the expression that has been
highlighted by the user, and any where\/let clauses associated with the
function.
-}

{-findPattern :: Term t => [PosToken] -- ^ The token stream for the
                                      -- file to be
                                      -- refactored.
                  -> (Int, Int) -- ^ The beginning position of the highlighting.
                  -> (Int, Int) -- ^ The end position of the highlighting.
                  -> t          -- ^ The abstract syntax tree.
                  -> (SrcLoc, PNT, FunctionPats, HsExpP, WhereDecls) -- ^ A tuple of,
                     -- (the function name, the list of arguments,
                     -- the expression highlighted, any where\/let clauses
```

```
                    -- associated with the function).
 -}
findPattern toks beginPos endPos t
  = fromMaybe (defaultPNT, defaultPat, error "Invalid pattern selected!")
              (applyTU (once_tdTU (failTU `adhocTU` inMatch
                                          `adhocTU` inCase
                                          `adhocTU` inDo
                                          `adhocTU` inList )) t)

    where
       --The selected sub-expression is in the rhs of a match
      inMatch (match@(HsMatch loc1  pnt pats rhs ds)::HsMatchP)
        -- is the highlighted region selecting a pattern?
        | inPat pats == Nothing = Nothing
        | otherwise = do
                        let pat = fromJust (inPat pats)
                        Just (pnt, pat, Match match)

      inCase (alt@(HsAlt s p rhs ds)::HsAltP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                         let pat = fromJust (inPat [p])
                         Just (defaultPNT, pat, MyAlt alt)

      inDo (inDo@(HsGenerator s p e rest)::HsStmtP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                         let pat = fromJust (inPat [p])
                         Just (defaultPNT, pat, MyDo inDo)
      inDo x = Nothing

      inList (inlist@(Exp (HsListComp (HsGenerator s p e rest)))::HsExpP)
        | inPat [p] == Nothing = Nothing
        | otherwise = do
                         let pat = fromJust (inPat [p])
                         Just (defaultPNT, pat, MyListComp inlist)
      inList x = Nothing

      inPat :: [HsPatP] -> Maybe HsPatP
      inPat [] = Nothing
      inPat (p:ps)
       = if p1 /= defaultPat
           then Just p1
           else inPat ps
              where
                p1 = locToLocalPat beginPos endPos toks p
```

Appendix H

Excerpts from the HaRe API

```
-----------------------------------------------------------------------------------------
-- | Same as 'hsVisiblePNs' except that the returned identifiers are in String format.
-- | Implemented by Huiqing Li.
hsVisibleNames:: (Term t1, Term t2, FindEntity t1, MonadPlus m) => t1 -> t2 -> m [String]
hsVisibleNames e t =do d<-hsVisiblePNs e t
                       return ((nub.map pNtoName) d)


{- | The Update class, implemented by Huiqing Li and Christopher Brown-}
class (Term t, Term t1)=>Update t t1 where

  -- | Update the occurrence of one syntax phrase in a given scope by another syntax phrase of the same type.
  update::(MonadPlus m, MonadState (([PosToken],Bool),(Int,Int)) m)
                         =>  t     -- ^ The syntax phrase to be updated.
                          -> t     -- ^ The new syntax phrase.
                          -> t1    -- ^ The contex where the old syntax phrase occurs.
                          -> m t1  -- ^ The result.

instance (Term t) =>Update RhsP t where
 update oldRhs newRhs t
   = applyTP (once_tdTP (failTP `adhocTP` inRhs)) t
   where
     inRhs (r::RhsP)
      | r == oldRhs && srcLocs r == srcLocs oldRhs
         = do (newRhs',_) <- updateToks oldRhs newRhs prettyprint
              return newRhs'
     inRhs r = mzero



instance (Term t) =>Update HsExpP t where

 update oldExp newExp  t
   = applyTP (once_tdTP (failTP `adhocTP` inExp)) t
   where
    inExp (e::HsExpP)
     | e == oldExp && srcLocs e == srcLocs oldExp
       = do (newExp', _) <-updateToks oldExp newExp prettyprint
            return newExp'
    inExp e = mzero

instance (Term t) =>Update HsStmtP t where

 update oldStmt newStmt  t
   = applyTP (once_tdTP (failTP `adhocTP` inStmt)) t
   where
    inStmt (s::HsStmtP)
     | s == oldStmt && srcLocs s == srcLocs oldStmt
       = do (newStmt', _) <-updateToks oldStmt newStmt prettyprint
```

```
            return newStmt'
    inStmt s = mzero

instance (Term t) => Update HsAltP t where
 update oldExp newExp  t
   = applyTP (once_tdTP (failTP `adhocTP` inExp)) t
   where
    inExp (e::HsAltP)
     | e == oldExp && srcLocs e == srcLocs oldExp
       = do (newExp', _) <-updateToks oldExp newExp prettyprint
            return newExp'
    inExp e = mzero

instance (Term t) =>Update PNT t where
  update oldExp newExp  t
   = applyTP (once_tdTP (failTP `adhocTP` inExp)) t
   where
    inExp (e::PNT)
     | e == oldExp && srcLocs e == srcLocs oldExp
       = do (newExp',_) <- updateToks oldExp newExp prettyprint
            return newExp'
    inExp e = mzero

instance (Term t) =>Update HsMatchP t where
  update oldExp newExp  t
   = applyTP (once_tdTP (failTP `adhocTP` inExp)) t
   where
    inExp (e::HsMatchP)
     | e == oldExp && srcLocs e == srcLocs oldExp
       = do (newExp',_) <- updateToks oldExp newExp prettyprint
            return newExp'
    inExp e = mzero

instance (Term t) =>Update HsPatP t where

 update oldPat newPat  t
   = applyTP (once_tdTP (failTP `adhocTP` inPat)) t
   where
    inPat (p::HsPatP)
     | p == oldPat && srcLocs p == srcLocs oldPat
       = do (newPat', _) <- updateToks [oldPat] [newPat] (prettyprintPatList False)
            return $ ghead "update" newPat'
    inPat e = mzero

instance (Term t) =>Update [HsPatP] t where

 update oldPat newPat  t
   = applyTP (once_tdTP (failTP `adhocTP` inPat)) t
   where
    inPat (p::[HsPatP])
     | sameOccurrence p oldPat
       = do  (newPat', _) <- updateToks oldPat newPat (prettyprintPatList False)
             return newPat'
    inPat e = mzero

instance (Term t) =>Update [HsDeclP] t where

 update oldDecl newDecl  t
   = applyTP (once_tdTP (failTP `adhocTP` inDecl)) t
   where
    inDecl (d::[HsDeclP])
     | sameOccurrence d oldDecl
       = do
            (newDecl',_) <- updateToks oldDecl newDecl prettyprint
            return newDecl'
    inDecl e = mzero

instance (Term t) =>Update HsDeclP t where
```

```
 update oldDecl newDecl  t
   = applyTP (once_tdTP (failTP `adhocTP` inDecl)) t
   where
    inDecl (d::HsDeclP)
     | sameOccurrence d oldDecl
       = do (newDecl',_) <- updateToks oldDecl newDecl prettyprint
            return newDecl'
    inDecl e = mzero

instance (Term t) =>Update HsImportDeclP t where

 update oldImpDecl newImpDecl  t
   = applyTP (once_tdTP (failTP `adhocTP` inDecl)) t
   where
    inDecl (d::HsImportDeclP)
     | sameOccurrence d oldImpDecl
       =do (newImpDecl', _) <-updateToks oldImpDecl newImpDecl prettyprint
           return newImpDecl'
    inDecl e = mzero

instance (Term t) => Update HsExportEntP t where
   update oldEnt@(EntE s) newEnt@(EntE s1) t
     = applyTP (once_tdTP (failTP `adhocTP` inEnt)) t
       where
         inEnt (e::HsExportEntP)
           | sameOccurrence e oldEnt
          =  do (s1',_) <- updateToks s s1 prettyprint
                return (EntE s1')
         inEnt e = mzero

instance (Term t) => Update HsTypeP t where
 update oldType newType t
   = applyTP (once_tdTP (failTP `adhocTP` inType)) t
   where
     inType (t::HsTypeP)
        | sameOccurrence t oldType
       = do (newType', _) <- updateToks oldType newType prettyprint
            return newType'
     inType t = mzero

instance (Term t) => Update HsConDeclP t where
 update oldType newType t
   = applyTP (once_tdTP (failTP `adhocTP` inType)) t
   where
     inType (t::HsConDeclP)
        | sameOccurrence t oldType
       = do (newType', _) <- updateToks oldType newType prettyprint
            return newType'
     inType t = mzero
```

Appendix I

The Stages of the Expression Processor

I.1 Stage 1 Parser

```
module Parser where
import Data.Char

data Expr = Literal Int | Bin Bin_Op Expr Expr
             | LetExp String Expr Expr | Var String
                deriving Show

data Bin_Op = Mul | Plus deriving Show

type Environment = [(String, Expr)]

addedMul = error "Added Mul Expr Expr to Expr"
addedBin = error "Added Bin Bin_Op Expr Expr to Expr"
addedLetExp = error "Added LetExp String Expr to Expr"

parseExpr :: String -> (Expr, String)
parseExpr (’ ’:xs) = parseExpr xs
parseExpr (’*’:xs) = parseBin Mul xs
parseExpr (’+’:xs) = parseBin Plus xs
parseExpr (x:xs)
     | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
               where
                 lit = parseInt xs
                 parseInt :: String -> String
                 parseInt [] = []
                 parseInt (x:xs) | isNumber x = x : parseInt xs
                                 | otherwise  = []
parseExpr (x:xs)
      | isChar x = case var of
```

```
                "let" -> (LetExp (isName name) expr1 expr2, rest3)
                x  ->  (Var var, remainder)
               where
                 name = parseVar remainder
                 (expr1, rest2) = parseExpr (drop (length name) remainder)
                 (expr2, rest3) = parseExpr rest2

                 isName xs | xs /= "let" = xs
                           | otherwise   = error "Parse Error"

                 remainder = (drop (length var) xs)
                 var = x: (parseVar xs)
                 parseVar :: String -> String
                 parseVar [] = []
                 parseVar (x:xs)  | isChar x = x : parseVar xs
                                  | otherwise = []
                 isChar :: Char -> Bool
                 isChar x = x `elem` ['a'..'z']

parseExpr xs = error "Parse Error!"

parseBin p_1 xs = (Bin p_1 parse1 parse2, rest2)
                        where
                          (parse1, rest1) = parseExpr xs
                          (parse2, rest2) = parseExpr rest1

eval :: Environment -> Expr -> Int
eval env (Literal x) = x
eval env (Bin op e1 e2) = eval_op op (eval env e1) (eval env e2)
eval env (LetExp n e e_2) = eval (addEnv n e env) e_2
eval env (Var n) = eval env (lookUp n env)

eval_op :: (Num a) => Bin_Op -> (a -> a -> a)
eval_op p_1@(Mul) = (*)
eval_op p_1@(Plus) = (+)
eval_op _ = error "Undefined Operation"

addEnv :: String -> Expr -> Environment -> Environment
addEnv name expr env = (name, expr) : env

lookUp :: String -> Environment -> Expr
lookUp name [] = error ("Cannot find variable "
           ++ name ++ " within the list of variables defined.")
lookUp name ((var, expr):xs)
      | name == var = expr
      | otherwise   = lookUp name xs
```

I.2 Stage 6 Parser

```
module Parser where
import Data.Char

data Expr = Literal Int | Bin Bin_Op Expr Expr
            | LetExp String Expr Expr | Var String
             deriving Show

data Bin_Op = Mul | Plus deriving Show

type Environment = [(String, Expr)]

addedMul = error "Added Mul Expr Expr to Expr"
addedBin = error "Added Bin Bin_Op Expr Expr to Expr"
addedLetExp = error "Added LetExp String Expr to Expr"

parseExpr :: String -> (Expr, String)
parseExpr (' ':xs) = parseExpr xs
parseExpr ('*':xs) = parseBin Mul xs
parseExpr ('+':xs) = parseBin Plus xs
parseExpr (x:xs)
  | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
               where
                 lit = parseInt xs
                 parseInt :: String -> String
                 parseInt [] = []
                 parseInt (x:xs) | isNumber x = x : parseInt xs
                                 | otherwise  = []
parseExpr (x:xs)
  | isChar x = case var of
                "let" -> (LetExp (isName name) expr1 expr2, rest3)
                x  ->  (Var var, remainder)
               where
                 name = parseVar remainder
                 (expr1, rest2) = parseExpr (drop (length name) remainder)
                 (expr2, rest3) = parseExpr rest2

                 isName xs | xs /= "let" = xs
                           | otherwise   = error "Parse Error"

                 remainder = (drop (length var) xs)
                 var = x: (parseVar xs)
                 parseVar :: String -> String
                 parseVar [] = []
                 parseVar (x:xs)  | isChar x = x : parseVar xs
                                  | otherwise = []
                 isChar :: Char -> Bool
                 isChar x = x `elem` ['a'..'z']
parseExpr xs = error "Parse Error!"

parseBin p_1 xs = (Bin p_1 parse1 parse2, rest2)
                        where
                          (parse1, rest1) = parseExpr xs
                          (parse2, rest2) = parseExpr rest1

eval :: Environment -> Expr -> (String, Int)
eval env (Literal x) = (show x, x)
eval env (Bin op e1 e2) = ((fst (eval_op op)) ++ " "
                           ++ (fst $ eval env e1) ++ " "
                           ++ (fst $ eval env e2),
                           (snd $ eval_op op) (snd $ eval env e1)
                           (snd $ eval env e2))
                   where
                      eval_op :: (Num a) => Bin_Op -> (String, (a -> a -> a))
                      eval_op p_1@(Mul) = ("*", (*))
                      eval_op p_1@(Plus) = ("+",(+))
                      eval_op _ = error "Undefined Operation"
eval env (LetExp n e e_2) = ("let " ++ n ++ " = "
                             ++ (fst $ eval env e) ++ " in "
                             ++ (fst $ eval env e_2),
                             snd $ eval (addEnv n e env) e_2)
eval env (Var n) = (n, snd $ eval env (lookUp n env))


addEnv :: String -> Expr -> Environment -> Environment
addEnv name expr env = (name, expr) : env

lookUp :: String -> Environment -> Expr
lookUp name [] = error ("Cannot find variable " ++ name ++
                  " within the list of variables defined.")
lookUp name ((var, expr):xs)
   | name == var = expr
   | otherwise   = lookUp name xs
```

I.3 Final Parser

```
module Parser where
import Data.Char

data Expr = Literal Int | Bin Bin_Op Expr Expr
             | LetExp String Expr Expr | Var String
             deriving Show

data Bin_Op = Mul | Plus deriving Show
```

```
type Environment = [(String, Expr)]

addedMul = error "Added Mul Expr Expr to Expr"
addedBin = error "Added Bin Bin_Op Expr Expr to Expr"
addedLetExp = error "Added LetExp String Expr to Expr"

parseVar :: String -> String
parseVar [] = []
parseVar (x:xs)  | isChar x = x : parseVar xs
                 | otherwise = []
isChar :: Char -> Bool
isChar x = x `elem` ['a'..'z']

parseInt :: String -> String
parseInt [] = []
parseInt (x:xs) | isNumber x = x : parseInt xs
                | otherwise  = []

isName xs | xs /= "let" = xs
          | otherwise   = error "Parse Error"

parse :: String -> Expr
parse input = parseFst input
   where
    parseFst :: String -> Expr
    parseFst (' ':xs) = parseFst xs
    parseFst ('*':xs) = parseBinFst Mul xs
    parseFst ('+':xs) = parseBinFst Plus xs
    parseFst (x:xs)
     | isNumber x = Literal (read (x:lit)::Int)
              where
                lit = parseInt xs
    parseFst (x:xs)
     | isChar x = case var of
               "let" -> LetExp (isName name) expr1 expr2
               x  ->  Var var
              where
                name = parseVar remainder
                (expr1, rest2) = parseExpr (drop (length name) remainder)
                expr2  = parseFst rest2

                remainder = (drop (length var) xs)
                var = x: (parseVar xs)
    parseBinFst p_1 xs = Bin p_1 parse1 parse2
                       where
                         (parse1, rest1) = parseExpr xs
```

```
                          parse2 = parseFst rest1


    parseExpr :: String -> (Expr, String)
    parseExpr (' ':xs) = parseExpr xs
    parseExpr ('*':xs) = parseBin Mul xs
    parseExpr ('+':xs) = parseBin Plus xs
    parseExpr (x:xs)
     | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
              where
                lit = parseInt xs

    parseExpr (x:xs)
     | isChar x = case var of
               "let" -> (LetExp (isName name) expr1 expr2, rest3)
               x  ->  (Var var, remainder)
              where
                name = parseVar remainder
                (expr1, rest2) = parseExpr (drop (length name) remainder)
                (expr2, rest3) = parseExpr rest2
                remainder = (drop (length var) xs)
                var = x: (parseVar xs)

    parseExpr xs = error "Parse Error!"

    parseBin p_1 xs = (Bin p_1 parse1 parse2, rest2)
                       where
                         (parse1, rest1) = parseExpr xs
                         (parse2, rest2) = parseExpr rest1

eval :: Environment -> Expr -> (String, Int)
eval env (Literal x) = (show x, x)
eval env (Bin op e1 e2) = ((fst (eval_op op)) ++ " "
                           ++ (fst $ eval env e1) ++ " "
                           ++ (fst $ eval env e2),
                           (snd $ eval_op op) (snd $ eval env e1)
                           (snd $ eval env e2))
                    where
                       eval_op :: (Num a) => Bin_Op -> (String, (a -> a -> a))
                       eval_op p_1@(Mul) = ("*", (*))
                       eval_op p_1@(Plus) = ("+",(+))
                       eval_op _ = error "Undefined Operation"
eval env (LetExp n e e_2) = ("let " ++ n ++ " = "
                             ++ (fst $ eval env e) ++ " in "
                             ++ (fst $ eval env e_2),
                             snd $ eval (addEnv n e env) e_2)
eval env (Var n) = (n, snd $ eval env (lookUp n env))
```

```
addEnv :: String -> Expr -> Environment -> Environment
addEnv name expr env = (name, expr) : env

lookUp :: String -> Environment -> Expr
lookUp name [] = error ("Cannot find variable " ++ name ++
                 " within the list of variables defined.")
lookUp name ((var, expr):xs)
   | name == var = expr
   | otherwise   = lookUp name xs
```

References

[1] PacSoft. Programatica: Integrating Programming, Properties and Validation. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`, 2005.

[2] Andreas Abel. Type-based termination of generic programs. *Science of Computer Programming*, 74(8):550–567, 2009. MPC'06 special issue.

[3] Miklós Ajtai and Yuri Gurevich. Datalog vs. first-order logic. *Journal of Computer Systems Science*, 49(3):562–588, 1994.

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[5] Roland Carl Backhouse and Paul F. Hoogendijk. Elements of a Relational Theory of Datatypes. In *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer, 1993.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on*

Reverse Engineering, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[7] F.L. Bauer, H. Ehler, R. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Vol. II: The Transformation System CIP-S, LNCS 292*, volume II. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1987.

[8] Friedrich L. Bauer, Rudolf Berghammer, Manfred Broy, Walter Dosch, Franz Geiselbrechtinger, Rupert Gnatz, E. Hangel, Wolfgang Hesse, Bernd Krieg-Brückner, Alfred Laut, Thomas Matzner, Bernhard Möller, Friederike Nickl, Helmuth Partsch, Peter Pepper, Klaus Samelson, Martin Wirsing, and Hans Wössner. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer, 1985.

[9] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Robert W. Bowdidge and William G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2):109–157, 1998.

[11] Jet Brains Inc. Intellij IDE - Java IDE With Refactoring Support. `http://www.jetbrains.com/idea/`, 2009.

[12] Ivan Bratko. *Prolog programming for artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[13] Christopher Brown and Simon Thompson. Refactorings that Split and Merge Programs. In *Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL*, September 2007.

[14] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

[15] Debra Cameron, James Elliott, and Marc Loy. *Learning GNU Emacs*. O'Reilly, December 2004.

[16] R Carlsson. Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3/doc/html`, 2007.

[17] Olaf Chitil. Source-Based Trace Exploration. In *IFL*, volume 3474 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2004.

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating Systems Errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[19] M. Cinneide and P. Nixon. Composite Refactorings for Java Programs. Technical report, Department of Computer Science, University College Dublin, 2000.

[20] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[21] The Haskell Committee. Haskell prime. `http://hackage.haskell.org/trac/haskell-prime/`, 2008.

[22] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[23] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.

[24] TclTK Developer Community. Tcl/tk. `http://www.tcl.tk/`, 2005.

[25] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification of the Haskell 98 Module System. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 17–28, New York, NY, USA, 2002. ACM.

[26] The DrIFT Development Team. The DrIFT homepage. `http://repetae.net/computer/haskell/DrIFT/`, 2003.

[27] Michael Ellims, James Bridges, and Darrel Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, March 2006.

[28] Eclipse Foundation. Eclipse - an Open Development Platform. `http://www.eclipse.org`, 2009.

[29] Eclipse Foundation. Eclipse Java Development Tools. `http://www.eclipse.org/jdt/`, 2009.

[30] NetBeans Foundation. Netbeans - an open-source ide. `http://www.netbeans.org`, 2009.

[31] NetBeans Foundation. Netbeans - refactoring. `http://refactoring.netbeans.org/`, 2009.

[32] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[33] Martin Fowler. Refactoring Home Page. `http://www.refactoring.com`, 2008.

[34] Alejandra Garrido and Ralph Johnson. Challenges of Refactoring C Programs. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 6–14, New York, NY, USA, 2002. ACM.

[35] Alejandra Garrido and Ralph Johnson. Refactoring C with Conditional Compilation. *Automated Software Engineering, International Conference on*, 0:323, 2003.

[36] Andy Gill. Introducing the Haskell Equational Reasoning Assistant. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 108–109. ACM Press, 2006.

[37] Andy Gill. A Haskell Hosted DSL for Writing Transformation Systems. Submitted to the IFIP Working Conference on Domain Specific Languages, December 2008.

[38] Andy Gill and Graham Hutton. The Worker Wrapper Transformation. *Journal of Functional Programming*, 2009.

[39] W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool Support for the Interactive Derivation of Formally Correct Functional Programs. Technical report, Universität Ulm Fakultät für Informatik, July 2002.

[40] Thomas Hallgren. Haskell Tools from the Programatica Project. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 103–106, New York, NY, USA, 2003. ACM Press.

[41] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[42] Johannes Henkel and Amer Diwan. Catchup!: Capturing and Replaying Refactorings to Support API Evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[43] Dean Herington. HUnit 1.0 Users Guide. `http://hunit.sourceforge.net`, 2002.

[44] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[45] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Survey*, 21(3):359–411, 1989.

[46] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer, 1995.

[47] Huiqing Li and Simon Thompson. Testing Erlang Refactorings with QuickCheck. In *Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007*, Freiburg, Germany, September 2007.

[48] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, 1996.

[49] Free Software Foundation INC. The C Preprocessor. `http://gcc.gnu.org/onlinedocs/cpp/`, 2008.

[50] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[51] J. Howard Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 171–183. IBM Press, 1993.

[52] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. pages 190–203. Springer-Verlag, 1985.

[53] Simon L. Peyton Jones and Ralf Lämmel. Scrap Your Boilerplate. In *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, page 357. Springer, 2003.

[54] Frederick P. Brooks Jr. The mythical man-month: After 20 years. *IEEE Software*, 12(5):57–60, 1995.

[55] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[56] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[57] Günter Kniesel and Helge Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[58] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297,

Oxford, 1970. Pergamon Press. Proceedings of a Conference held at Oxford under the auspices of the Science Research Council Atlas Computer Laboratory, 1967.

[59] Raghavan Komondoor and Susan Horwitz. Tool Demonstration: Finding Duplicated Code Using Program Dependences. In *ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 383–386. Springer, 2001.

[60] W. Korman and W. Griswold. Elbereth: Tool Support for Refactoring Java Programs. Master's thesis, University of California, San Diego Department of Computer Science and Engineering, May 1998.

[61] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg. Use cases for refactoring in erlang. In *CEFP*, volume 5161 of *Lecture Notes in Computer Science*, pages 250–285. Springer, 2007.

[62] Ralf Lämmel and Simon Peyton Jones. Scrap More Boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[63] Ralf Lämmel, Simon Thompson, and Markus Kaiser. Programming errors in traversal programs over structured data. In *8th Workshop on Language Description, Tools and Applications*, ENTCS. Springer, April 2008.

[64] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In

PADL, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 2002.

[65] Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[66] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[67] Huiqing Li. *Refactoring Haskell Programs.* PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2006.

[68] Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In *ACM SIGPLAN 2003 Haskell Workshop*, pages 27–38. Association for Computing Machinery, August 2003.

[69] Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE, September 2006.

[70] Huiqing Li and Simon Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *PEPM*, pages 169–178. ACM, 2009.

[71] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer:

HaRe, and its API. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science.

[72] Zhenmin Li, Shan Lu, and Suvda Myagmar. Cp-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006. Member-Yuanyuan Zhou.

[73] Jesse Liberty. *Learning C#.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[74] Chamond Liu. *Smalltalk, objects, and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[75] Simon Marlow. Hatchet: A Type Checking and Inference Tool for Haskell 98. `http://www.cs.mu.oz.au/ bjpop/code.html`, 2007.

[76] Simon Marlow. Haddock: A Haskell Documentation Tool. `http://www.haskell.org/haddock/`, 2008.

[77] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2009.

[78] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2005.

[79] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2002.

[80] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[81] Sun Microsystems. MySQL. `http://www.mysql.com/`, 2008.

[82] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[83] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *OOPSLA*, pages 235–250, 1996.

[84] Alan Mycroft. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 1980.

[85] Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134, New York, NY, USA, 2004. ACM Press.

[86] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[87] Steve Oualine. *Vim (Vi Improved)*. Sams, April 2001.

[88] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

[89] Helmut A. Partsch. *Specification and transformation of programs: a formal approach to software development.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[90] Alberto Pettorossi. A Powerful Strategy for Deriving Efficient Programs by Transformation. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 273–281, New York, NY, USA, 1984. ACM.

[91] Simon Peyton Jones. A Pretty Printing Library in Haskell, Version 3.0. 1997. `http://research.microsoft.com/Users/simonpj/downloads/prettyprinter/pretty.html` .

[92] Simon Peyton Jones and Kevin Hammond. *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, December 2003.

[93] Simon Peyton Jones and John Hughes. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. `http://haskell.org/onlinereport/`.

[94] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[95] The Refactor-fp Group. The Haskell Editing Survey. `http://www.cs.kent.ac.uk/projects/refactor-fp/surveys/haskell-editors-July-2002.txt`, 2004.

[96] The Refactor-fp Group. Refactoring Functional Programs. `http://www.cs.kent.ac.uk/projects/refactor-fp`, 2008.

[97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[98] Donald Roberts. Practical Analysis for Refactoring. Technical report, Champaign, IL, USA, 1999.

[99] N. Rodrigues and L. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.

[100] Niklas Röjemo. Highlights from nhc—a space-efficient haskell compiler. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 282–292, New York, NY, USA, 1995. ACM.

[101] C.H. Roy and R. Cordy. A survey on software clone detection research. Technical report, School of Computing, Queen's University at Kingston, Canada, 2007.

[102] Chris Ryder and Simon Thompson. Porting HaRe to the GHC API. Technical Report 8-05, Computing Laboratory, University of Kent, Canterbury, Kent, UK, October 2005.

[103] Tim Sheard. Generic Unification via Two-level Types and Parameterized Modules. volume 36, pages 86–97, New York, NY, USA, 2001. ACM.

[104] Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 157–166, New York, NY, USA, 2006. ACM Press.

[105] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, 1990.

[106] Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, New York, NY, USA, 1995. ACM.

[107] Simon Thompson. Higher-order + Polymorphic = Reusable. Technical report, University of Kent, Canterbury, Kent, UK, May 1997.

[108] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 2 edition, 1999.

[109] F. Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[110] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. *SIGPLAN Not.*, 38(11):13–26, 2003.

[111] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, pages 174–, 1999.

[112] Mark Anders Tullsen. *Path, a program transformation system for Haskell.* PhD thesis, Computer Science, Yale University, New Haven, CT, USA, 2002. Director-Paul Hudak.

[113] David Ungar and Randall B. Smith. Self: the power of simplicity. In *Proceedings of OOPSLA*, 1987.

[114] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[115] E Visser. Stratego: Strategies for Program Transformation. Technical report, Universiteit Utrecht, 2003.

[116] Eelco Visser. HsOpt: An Optimizer for the Helium Compiler. Technical report, Universiteit Utrecht, 2003. `http://www.stratego-language.org/twiki/bin/view/Stratego/HsOpt`.

[117] Eelco Visser. Hsx: A Framework for Haskell Transformation. Technical report, Universiteit Utrecht, 2003. `http://www.statego-language.org/twiki/bin/view/Stratego/HSX`.

[118] Martin Ward and Keith Bennett. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering*, 5(1):25–47, 1995.

[119] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[120] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979.