Analysis-based Refactorings for Haskell

Christopher Brown
Computing Laboratory
University of Kent
United Kingdom
cmb21@kent.ac.uk

Simon Thompson
Computing Laboratory
University of Kent
United Kindgom
S.J.Thompson@kent.ac.uk

ABSTRACT

Refactoring is the process of improving the design of existing programs without changing their external behaviour. Refactoring can make a program easier to understand or modify if applied appropriately. Preserving behaviour guarantees that refactoring does not introduce (or remove) any bugs. Refactoring has taken a prominent place in software development and maintenance, but most of the recent success has been in the OO and XP communities.

HaRe, the Haskell Refactorer, developed at the University of Kent by Huiqing Li, Claus Reinke and Simon Thompson, is a refactoring tool for Haskell 98. This paper presents a number of new refactorings for HaRe that fall under the category of program-analysis and it also presents some new refactorings that make use of the GHC type-checker; the new architecture for HaRe encompassing the type-checker is also discussed.

General Terms

Transformation, Refactoring, Program Slicing, Languages, Design

1. INTRODUCTION

Consider the following piece of Haskell code:

```
f :: [Int] -> [Int]
f list = h list
       where
          h [] = []
          h (x:xs) = (+1) x : h xs
```

We have another function `g` such that `f` and `g` both use a local definition that shares the same functionality of taking a list and applying something to every element:

```
g :: [Char] -> [Char]
g list = g list
       where
          h [] = []
          h (x:xs) = toUpper x : h xs
```

We decide that `h` can be promoted to the top level, and that by adding a parameter to it, the functionality of `(+1)` and `toUpper` can be passed to it depending on the context. Renaming `h` to something useful like `apply` leads to the following program:

```
f :: [Int] -> [Int]
f list = apply (+1) list

g :: [Char] -> [Char]
g list = apply toUpper list

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

Clearly `apply` is in fact just `map`, so we rename all calls to `apply` with `map` and delete the definition of `apply`.

It is common practice to write a program and then discover that small structural changes can simplify the program. This process is called 'refactoring' [10]: the process of changing the structure of a computer program without altering its behaviour. The focus on structural changes rather than changes in functionality distinguish refactoring from general 'code meddling'. A structural change can potentially make a function simpler, by removing duplicate code, say, or can be a preparatory step for an upgrade or extension of a system.

HaRe [8, 15, 7] currently works in (x)Emacs and Vim and supports a wide range of refactorings, such as renaming, promoting and generalising a definition for example. HaRe uses the Programatica [12] front-end to do all the parsing and language manipulation. Haskell is used as the language to create the refactorings and also as the language to refactor.

Until now, refactorings added to HaRe did not need type information. Haskell being a strongly typed language means that there are a number of refactorings and transformations that require the use of type information. This paper will present a number of new refactorings for HaRe that make use of type information and also some ideas for new refactorings that require type information. The paper also proposes an evaluation of the current HaRe refactorings: in some cases it may be necessary to modify some of the HaRe refactorings to make them type safe.

The paper is divided into a number of sections: firstly we begin with a brief overview of the Haskell refactorer HaRe.

Secondly we introduce a number of new refactorings for HaRe, some of the refactorings mentioned introduce a requirement for type information. We then talk about the architecture of HaRe and our motivation for using the GHC API [2]. Finally we propose a number of transformations and refactorings for HaRe. Some of the refactorings and transformations mention build upon existing work presented in this paper and are new ideas; a refactoring calculus is also proposed.

2. HARE: THE HASKELL REFACTORER

HaRe currently has support for:

- The full Haskell 98 standard. Although there has been an attempt by Chris Ryder [13] to port HaRe to the to the *de facto* GHC [1] Haskell standard;
- Working within programmer's existing tools. HaRe currently works in (x)Emacs and Vim. Rather than being a stand-alone tool, HaRe allows programmers to augment their existing practice with zero overhead;
- Preserving the layout of source programs. Although layout is significant in Haskell, there are many different layout styles adopted by different programmers. In most cases having code reformatted using a 'pretty printer' would be totally unacceptable.
- Project aware refactorings. All refactorings in Haskell work within the scope of the entire project, incorporating the namespace of all the modules imported (apart from the prelude library).

The following is an example of some of the refactorings that HaRe supports:

- **Rename** a definition. All occurrences of calls to the definition are replaced with calls to the new definition name.
- **Promote** or **Demote** a definition. Lift a local definition to the top level, or demote a top-level definition to a local scope. HaRe checks that the identifier does not conflict with other definitions of the same name after the demotion or promotion.
- **Generalise** a definition. Higher order functions allow specific functionality to be abstracted into a function, which can then be passed as a parameter.
- **Folding** function definitions. Replace duplicate expressions with calls to a function definition encompassing the functionality of the expression.

Using a refactoring tool such as HaRe allows programmers to take a much more exploratory and speculative approach to design. Large-scale refactorings can be accomplished in a single step and can also be undone without any effort.

3. ANALYSIS-BASED REFACTORINGS

There have been a number of new refactorings added to HaRe since the last official release. Some of these new refactorings require program analysis for their correct implementation and are discussed in this section.

In this section we present a number of new refactorings that have been developed for HaRe:

- **Adding a constructor:** add a new constructor to a data type and automatically generate new clauses to patterns that reference the data type.
- **Remove redundant declarations:** select a function, and remove any definitions defined locally in the function that are not needed to compute the right hand side.
- **Program slicing:** highlight a sub-expression and create a new definition encompassing all the names and expressions that are needed to compute the sub-expression.
- **Splitting a definition:** select a definition that returns a tuple. Create new definitions encapsulating the functionality of each tuple element.
- **Folding expressions against function definitions:** highlight a function where the right hand side is an expression. Replace all occurrences of the sub-expression within the program with a call to the function, in some cases passing in literals as arguments to the function.
- **Convert a data type into a newtype**: Select a data type with a single unary constructor and convert it into a corresponding `newtype` representation. This refactoring has the potential of changing the strictness properties within a program.

HaRe has recently undergone an engineering process to allow refactorings to gain access to type information. Some of the refactorings presented in this section use the type checker. The motivation of extending the refactorings to use type information is described in the cases where it is used.

3.1 Adding a Constructor

Adding a constructor to a data type may seem like a trivial program transformation, but it leads to an interesting problem in Haskell. It is often common practice when developing new programs in Haskell, to start with a small data definition, and then add to that data definition to increase the program complexity in an incremental fashion. In Haskell, programmers often create data types to model a system and then create functions to perform some analysis or transformation on that model. Pattern matches are added to functions that work over the data type to capture different behaviors. When a new constructor is added to a data type, new pattern clauses are often also added to function definitions to specify the behavior of the function for the new constructor. This process can be done automatically in a refactoring.

Let us consider a simple programming problem as a proof of concept. We are presented with a simple programming

```
data AST = Program Exp
data Expr = Num Int

eval :: Expr -> Int
eval (Num a) = a

add :: Expr -> Expr -> Int
add (Num a) (Num b) = a + b
```

Figure 1: The Initial Program

```
data AST = Program Exp
data Expr = Num Int  |  Add Expr Expr

eval (Num a) = a

add :: Expr -> Expr -> Int
add (Num a) (Num b) = a + b
```

Figure 2: Adding a Constructor Refactoring 1

```
data AST = Program Exp
data Expr = Num Int  |  Add Expr Expr

addedAdd = error "added Add Expr Expr to Expr"

eval (Num a) = ...
eval (Add a b) = a `add` b

add (Num a) (Num b) = ...
add (Add a b) (Add c d) = addedAdd
add (Num a) (Add b c) = addedAdd
add (Add a b) (Num c) = addedAdd
```

Figure 3: Adding a Constructor Refactoring 2

language, and consider creating a Haskell program to evaluate the language (a simple interpreter). It makes sense to represent the language in Haskell, using a data type to represent the language syntax; and creating functions to evaluate the terms represented within the data type. Starting with simple cases, we incrementally add more complexity to our evaluator to handle increasingly more complex terms. Here is an example of two distinct terms in our language separated by semi-colons:

```
 1;
 1 + 2 - 3 ;
```

The first expression is the number `1`; the second is a more complicated expression showing addition and subtraction. It makes sense to model the use of the addition and subtraction operators at the abstract syntax tree (AST) level.

Consider Figure 1. A new data type to represent the AST for the programming language has been defined. Currently the AST only has provision for a single kind of expression: a number (defined as `Num Int`). The program also has some simple evaluation functions that work over the AST: `eval` to evaluate an expression and `add` to represent the functionality to add two expressions together.

The programmer has now decided to represent addition at the AST level, therefore requiring a new constructor `Add` to be added taking two `Expr`s as parameters.

This produces the program shown in Figure 2. However, the result of calling `add` with the following parameters will lead in a runtime error:

```
-- (1+2) + (3 + 4)
add (Add 1 2) (Add 3 4)
```

The function `add` only accepts two numbers. Clearly new pattern match clauses need to be added to the definition of `add`. The refactoring can do this automatically for us, resulting in the program in Figure 3. The new patterns added to `eval` and `add` cover every possible scenario. The right hand sides of the new pattern clauses call the function `addedAdd`, which displays a message indicating that a new constructor has been added to the data type `Expr`. This is necessary because we cannot make assumptions of the intention of the new constructor.

3.1.1 Using Type Information

The transformation described in Figure 3 was a naive implementation. It simply looked for any occurrence of a Constructor name in a pattern, and then attempted to add new clauses to the patterns that reference the data type in question. We say this is naive because there are a number of cases where the refactoring can overlook situations where new clauses should be added to patterns, but the constructors in the data type are not directly referenced in the pattern. A simple example of this is in Figure 4. Here a function

```
data AST = PRogram Expr
data Expr = Num Int | Add Expr Expr

left :: Expr -> Expr -> Expr
left a b = a
```

Figure 4: A Simple Program

```
data AST = PRogram Expr
data Expr = Num Int | Add Expr Expr | Sub Expr Expr

addedSub = error "Added Sub Expr Expr to Expr"

left :: Expr -> Expr -> Expr
left (Sub a b) c = addedSub
left a (Sub b c) = addedSub
left a b = a
```

Figure 5: A Simple Program with an added constructor

`left` is declared, which takes two `Expr` types as parameters and returns an `Expr` value. The only definition of `left` simply gives back the first `Expr` parameter. Now, suppose that a new constructor is added to `Expr`, called `Sub` (taking two `Int`s as parameters) as shown in Figure 5. It may seem that new clauses to capture the new constructor `Sub` in both arguments of `left` are not needed, as there is a catch all clause (denoted by `left a b = a`). The refactorer cannot assume what the intended behavior for the new constructor is going to be, so adds new clauses to `left` to capture the new constructor, and places a call to a function which returns an error message on the right hand side. In this case, strictly speaking the refactoring does not need type information, as it can simply look at the type signature of the function. But there are some cases where the function has no type signature, but the inferred type is of the data type in question.

The program in Figure 6 shows a simple data type with two constructors, `AA` and `BB`. The type of `f` is inferred by the type checker to have `f :: Data -> Data` (`f` calls `g`, which takes as a parameter a `Data` type). The program in Figure 7 shows the result of adding a new constructor `CC` to the data type `Data`. The refactoring has placed new clauses for definitions `f` and `g` to capture the undetermined intention of the new constructor by displaying an error.

Interestingly, although adding a constructor is labelled as a refactoring, it is in fact a transformation. There is a distinct difference between a refactoring and a transformation that

```
data Data = AA | BB
f x = g x

g AA = AA
g BB = BB
```

Figure 6: A simple program

```
data Data = AA | BB | CC
addedCC = error "added CC to Data"

f CC = addedCC
f x = g x

g AA = AA
g BB = BB
g CC = addedCC
```

Figure 7: Adding a constructor to `Data`

```
h :: (Int, Int)
h = result 2
      where
        f = 42
        result 2 = (1,2)
        result x = res
        res = (23, 24)
        g = 44
```

Figure 8: A simple function with some redundant declarations

might affect the behavior of the program. The top level behavior of a program could change if new pattern matches are added to a function and those functions are then called with the new constructor as a parameter. Adding new patterns to the functions in this way can cause the parameters to become strict. However, the local behavior of the function is preserved. Figure 7 shows a pattern clause added to `f` to capture the constructor `CC`. The behavior is still preserved as long as the constructor `CC` is not used.

3.2 Remove Redundant Declarations

This refactoring removes all the unused declarations in the local scope of a selected function. This refactoring is also used in the two refactorings that follow: program slicing and splitting a definition.

Figure 8 is an example of how this refactoring is used. The refactoring then cleans `h` by removing from it any local definitions that are needed to compute the expression `result 2`.

The refactoring works by building a small data dependence graph of the definitions needed to compute the expression on the right hand side of the selected function. By following the path of any function calls, it can be seen that: `result` calls

```
h :: (Int, Int)
h = result 2
      where
        result 2 = (1,2)
```

Figure 9: A simple function

res. The definitions f and g are needed to compute result or res. If any of the function makes a call to a definition in another scope, the refactoring simply ignores it.

It is quite common to change a function and then be left over with local definitions that are no longer needed. A refactoring to remove these definitions proves useful.

3.3 Program Slicing

3.3.1 Slicing Imperative Programs

A useful program transformation technique known as *program slicing* has been implemented as a refactoring for HaRe. Program Slicing [17] is a method for decomposing programs by analyzing their data and control flow. A program slice consists of the parts of a program that potentially effect the values computed at some point of interest, called a slicing criterion [16]. Usually, a slicing criterion consists of a pair (line-number, variable). The program slice with respect to some criterion are the parts of the program that have a direct or indirect effect on computing the values at the slicing criterion. Program slices are usually computed from a *Program Dependance Graph* [4] that makes explicit both the data and control dependencies for each operation in a program.

The first set of program slicing algorithms took a backwards slice of the criterion. A backwards slice is all parts of the program that had an effect on the criterion in question. Another form of program slicing is a forwards slice, starting with the program criterion, or the program point of interest a forwards slice is all parts of the program that the criterion will effect. There are two types of slice: *static* program slicing, which means that all possible computations of a program are considered and *dynamic*program slicing, which considers only particular computations of interest (i.e if some of the program's input is known in advance).

3.3.2 Slicing Functional Programs

There has been little work on program slicing for Haskell. Silva et al. [11] introduces a dynamic slicing technique for a lazy functional logic language. The Haskell debugger, Hat [3] also includes a program slicer. However, there is no independent program slicing tool available for Haskell. Therefore our motivation was to create a program slicing transformation tool that is implemented as a refactoring in HaRe.

The meaning of program slicing in the context of Haskell is somewhat different to that in an imperative program. In Haskell, pure functions do not produce side-effects, so performing a program slice on a particular function is trivial: a simple pruning of the call graph is needed. It makes more sense to select a particular expression or a structure within the program (such as a tuple or a list) and then produce a program consisting of all names and expressions that are needed to compute the expression of interest, or compute each element of a tuple. This is the approach that we took.

Our program slicer works by creating a data dependance graph based on a particular sub-expression of interest, and then removing parts of the definition that are not required to compute the names stored in the graph. Our program slicer only looks under the scope of the particular definition of interest, and does not compute a program slice based on the entire program.

```
f x y = let result = x + y ;
            result3 = z + 23 in result + result3
            where
               result2 = h 12 z
               z = 23
               h = 12
```

Figure 10: A simple program

```
f x y = let result = x + y + result2 in result
             where
               result2 = h 12 z
               z = 23
```

Figure 11: A program slice for expression `result`

The program slicer for HaRe requires the selection of a particular sub-expression of interest and then selecting the refactoring from the HaRe menu. The refactoring then creates a new definition with the right hand side being the selected sub expression. Any local definitions required to compute the sub-expression are added to the new definition within a where or let clause respectively.

Consider the program in Figure 10. Suppose the programmer is only interested in the parts of f that affect the computation of the expression result. The programmer highlights the expression result and selects the refactoring from the HaRe menu. HaRe then produces a new definition shown in Figure 11.

The refactoring is very useful when the programmer wishes to extract a certain piece of functionality from a definition and then expand on that functionality, creating a new definition in the process.

3.4 Splitting a Definition

A function can return two values: for instance, the library function splitAt returns the elements of a list before a specified index as well as the elements of a list after the index. Sometimes we want to extract the two distinct operations and move them into separate functions. Building upon the program slicing refactoring presented in the previous sec-

```
f :: Int -> (Int, Int)
f 1 = (1 , 2)
f x = (x , x)
```

Figure 12: A definition that returns a tuple

```
f1 :: Int -> Int
f1 1 = 1
f1 x = x

f2 :: Int -> Int
f2 1 = 2
f2 x = x
```

Figure 13: Splitting the functionality of a tuple into separate definitions

tion, we present another refactoring for HaRe that allows a function returning a tuple to be split into separate definitions: encapsulating the functionality of each tuple element.

Consider the program in Figure 12. Selecting the definition `f` and choosing the "slicing over tuples" refactoring from the HaRe menu, the program slicer gives back two new definitions `f1` and `f2`, as shown in Figure 13.

The refactoring works by continually calling the program slicing refactoring on each element of the tuple. Each time the program slicer is called a new definition is created. In the cases where the right hand side of the selected definition is not an explicit tuple (a function call, say) then the refactoring traverses to the function definition that is being called (if it is a local definition) and continues the process. The refactoring also takes into consideration function matches (multiple definitions of a function), and performs a separate slice on each match.

3.4.1 Using Type Information

There are situations where it becomes difficult to determine whether one has selected a function that returns a tuple. For example the right hand side of a selected function may be an explicit tuple as in:

```
f :: (Int, Int)
f = (1 , 2)
```

In this case it is easy to split the definition and type information is not needed. However in cases where the right hand side is not an explicit tuple, but a function application then the process becomes more difficult. For example:

```
f = result
      where
        result = (1, 2)
```

In this case it is still possible to check without using the type checker that the function returns a tuple, but to do so would require traversing the AST until `result` is found and then checking to see if the right hand side of `result` is an explicit tuple. Instead it may be a call to a function returning a tuple, such as `splitAt`. In cases like this, it may not be possible to determine that the function returns a tuple, without doing lots of AST traversals. Using type information makes the process much simpler.

In the case where the right hand side of the function is not an explicit tuple, but a function call say, then the type checker is called to find the type of the expression on the right hand side. If the type of the expression is a tuple then the program slicer jumps to the definition of the function that is being called and the process is repeated. Once an explicit tuple has been found in the chain, the program slicer can create new definitions for slices of each tuple element. If the function calls are all local definitions, the right hand side of the function is preserved, but the local definitions change to return single elements instead of tuples.

```
addOne :: Int -> Int
addOne n = 1 + n

mulTwo :: Int -> Int
mulTwo n = 2 * n

addTwo :: Int -> Int -> Int
addTwo n m = n + m
```

Figure 14: A Simple Program showing some duplicate expressions

```
addOne n = addTwo 1 n

...

addTwo n m = n + m
```

Figure 15: Folding against `addOne`

3.5 Folding expressions

This refactoring was implemented as an attempt to help reduce the amount of duplicate code within a program. The user selects a particular function of interest with, what the user believes to be, a common sub-expression on the right hand side, and then selects the refactoring from the HaRe menu. HaRe then attempts to match the expression in the selected function with all occurrences of the expression within the program. In particular, HaRe looks for expressions that have the same partial structure as the expression in the selected function. HaRe then replaces the expression with a call to the selected function.

To show how this works consider the example in Figure 14. The functions `addOne` and `addTwo` share the same partial structure: they both add two `Int`s together. Suppose the user has realised this and selects the function `addTwo`, HaRe then looks for the `n + m` shape throughout the program, and replaces all matches with a call to `addTwo` giving any literals or identifiers as arguments. The result is shown in Figure 15.

The refactoring follows a naive approach to matching partial structures. If, for example, the function `addOne` in Figure 14 was defined as follows:

```
addOne :: Int -> Int
addOne n = (+) 1 n
```

Then the refactoring would not replace the expression `(+) 1 n` with a function call to `addTwo`. If a function application is infix, then it is represented in a different way to a normal application in the Programatica AST. Additional work could be done to correct this, but the refactoring serves as an example of a way to approach the problem of reducing duplicate code in a large Haskell project.

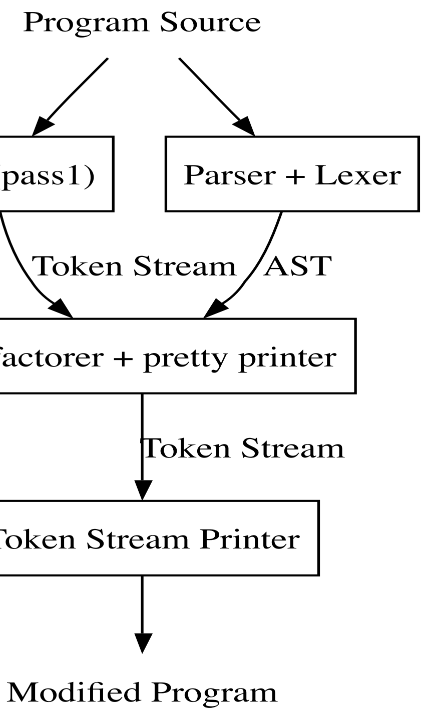

Figure 16: The implementation of HaRe

3.6 Converting a Data Type into a Newtype

This refactoring simply converts a selected data type (with a unary constructor) into a newtype:

```
data New = Con1 Int
```

Is converted to:

```
newtype New = Con1 Int
```

There are some implications for converting a data type into a newtype [14]. Converting a data type into a new type is really a Haskell *transformation* rather than a *refactoring*, as the behaviour of the program will change. Currently, the refactoring issues a warning to the user before the refactoring is applied clearly stating that the program properties, including strictness, will change after the transformation. As it stands, this refactoring is relatively trivial, but could be extended to alter the structure of the program further so that the behavior of the program is unchanged.

4. USING THE TYPE CHECKER

The current release of HaRe uses Programatica's lexer [5] and parser. Figure 16 shows a graphical overview of the implementation architecture of HaRe. To perform a refactoring, the parser takes the program source and passes it into the lexer to create a token stream and also to the parser to create the AST. The AST is used only as an auxiliary representation of the program to guide the direct modification of the token stream. The refactorer performs program analysis and transformation on the AST. Once the AST is modified, the refactorer also modifies the token stream to reflect the changes in the AST. The token stream needs adjustment to counteract the side-effects of the transformation on the layout rules.

HaRe attempts to preserve the layout of the source programs. Instead of using a pretty printer to present a modified AST to the programmer in a concrete form, the new program is extracted from the token stream. Preserving both comments and layout style for most programs. Some refactorings require new code to produced, in which case no layout information can be inferred so the pretty printer is used.

In order to create refactorings that used type information we originally made use of the Programatica type-checker. Programatica allows two methods of parsing a Haskell program, where both methods return an AST. Straightforward parsing returns an AST representing the names and expressions that occur within the program. Type checking also returns an AST, but in addition to names and expressions, the AST is also decorated with type information, giving types for every name and expression within the program. The type decorated AST also contains other information to help the type checker deal with over loading. A dictionary is used for passing the implementation of type classes as parameters to functions that use over loaded functions.

However, the Programatica type checker proved to be slow in practice, so instead, we integrated HaRe with the GHC API. HaRe still uses the Programatica front-end, to parse the source program, but also utilises the type-checker from GHC to get type information. An attempt was made by Chris Ryder to port HaRe to the GHC API, but due to time constraints the port was not fully realised. When considering using the GHC API for type checking, we decided to not re-engineer HaRe, but instead to only call the GHC type checker when needed, simply calling it with an arbitrary expression to gather type information.

5. FUTURE WORK AND CONCLUSIONS

The ideas presented in this paper have shown how Haskell is a complex language, and that simple transformations usually require detailed program analyses. Engineering HaRe to use type information has opened up a whole world of more complex transformations to be added to HaRe that require type information. One example is adding or removing class instances: such a refactoring would have to check that removing an instance does not interfere with any functions that rely on that class in a context. Similarly, adding an instance to a class could mean that some functionality would have to be provided to the new instance to cope with existing functions that use the overloaded operations of a class.

5.1 Merging a definition

A very interesting refactoring is the converse of splitting a definition. Merging definitions together to form a new definition which returns a tuple: each element of the tuple

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 xs = []
take n (x:xs) = x : ls
                 where
                    ls = take (n-1) xs

drop :: Int -> [a] -> [a]
drop n [] = []
drop 0 xs = xs
drop n (x:xs) = rs
                  where
                    rs = drop (n-1) xs
```

Figure 17: take and drop

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n [] = ([], [])
splitAt 0 xs = ([], xs)
splitAt n (x:xs) = (x:ls, rs)
                    where
                       (ls, rs) = splitAt (n-1) xs
```

Figure 18: take and drop merged together

being the computation of one of the definitions in the merge. This refactoring is particularly useful to compose functions together. Consider the two functions in Figure 17, selecting these definitions and choosing the merge refactoring, the function `splitAt` is introduced, shown in Figure 18.

Together with splitting a definition, merging and splitting allow the user to split a definition into separate subcomponents, alter the functionality of one of the sub components or even introduce new functionality and then merge back together to form the original definition (with new functionality).

5.2 Generalising a data type

The refactoring to add a constructor to a data type was discussed, it is also possible to extend this idea further to generalise a data type. An example of this would be a data type that represented a tree in Haskell:

```
data Tree = Node Int | Leaf Tree Tree

succ :: Tree -> Tree
succ (Node x) = x + 1
succ (Leaf x y) = Leaf (succ x) (succ y)

...

transformTree :: Tree -> Tree
transformTree = ... succ ...
```

By giving the data type `Tree` a general type, it is possible to then generalise the functionality of `succ` and `transformTree`:

```
h l = (ls, w)
      where
          ls = take w l
          rs = length l
          w = rs - 1
```

Figure 19: A Function Before Splitting

```
f1 = ls
     where
       ls = take (rs - 1) l
       rs = length l

f2 = rs - 1
    where
       rs = length l

h l = ...
```

Figure 20: A Function After Splitting

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

succ :: Tree a -> Tree a
succ f (Node x) = f x
succ f (Leaf x y) = Leaf (f x) (f y)

...

transformTree :: Tree Int -> Tree Int
transformTree = ... succ (+1) ...
```

This refactoring could also be an example of where composing refactoringsproves useful. Introducing a new refactoring to generalise a data type and then composing this with the generalisation refactoring.

5.3 A Symbolic Evaluator

Splitting a definition could be extended further to optimise code, by symbolically evaluating expressions or removing definitions defined in a where clause. Consider the function in Figure 19, splitting the definition into two separate functions and then performing some symbolic evaluating and optimisation produces the program in Figure 20.

5.4 Refactoring HaRe

As mentioned previously the refactorings in the current release of HaRe do not use type information. It would be interesting to conduct an analysis of HaRe and how its refactorings can produce untypeable code, by using the type checker the refactorings could be made type safe.

Examples of refactorings that would benefit from type information are:

- **Generalisation**: Currently the refactoring to generalise a definition comments out the type signature, it would be better practice to use the type-checker to infer the new type of the transformed definition.

- **Promotion** and **Demotion**. Promoting a polymorphic function to a top level, say, and removing its arguments: making it monomorphic might impose a monomorphism restriction. The same can be said for demoting a definition.

5.5 A Refactoring Calculus

When implementing many of the refactorings presented in this paper, we usually take the same preparatory steps. First we make decisions on what we believe the behavior of the refactoring should be (our opinions do not in all cases agree with the opinions of others); secondly we implement the refactoring and finally we test the refactoring until we are certain it produces the desired result for our case analyses.

A much more elegant way to create a refactoring would be to use a *refactoring calculus*. Such a calculus would allow one to completely specify an entire refactoring using mathematics. A refactoring calculus would need a way to specify any *pre* and *post* conditions that would need to be satisfied. In order to check that the refactoring guarantees behavior, the calculus would need a way to allow one to easily reason about their refactoring.

It would be interesting to conduct a *taxonomy of refactorings*, finding out exactly what is essential to create a refactoring for Haskell; and what is the commonality between all the refactorings implemented so far. Perhaps there is a more elegant solution for specifying and implementing these refactorings. Creating a calculus would allow the programmer to define refactorings including the behavior expected from the refactoring. There has been some work on formalising refactorings for Haskell by Li [6] and Cornélio [9], but to our knowledge no work has been done to allow the complete specification of a refactoring.

6. REFERENCES

[1] The Glasgow Haskell Compiler. http://www.haskell.org/ghc, 2006.

[2] K. Angelov and S. Marlow. Visual haskell: A full-featured haskell development environment. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 5–16. ACM Press, September 2005.

[3] O. Chitil. Source-based trace exploration. In *Draft Proceedings of the 16th International Workshop on Implementation of Functional Languages, IFL 2004*, pages 239–244. Technical Report 0408, University of Kiel, September 2004.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[5] T. Hallgren. A Lexer for Haskell in Haskell. http://www.cse.ogi.edu/ hallgren/Talks/LHiH/2002-01-14.html, 2002.

[6] H. Li. *Refactoring Haskell Programs.* PhD thesis, Computer Science, University of Kent, 2006.

[7] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In J. Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop.* Association for Computing Machinery, August 2003. ISBN 1-58113-758-3.

[8] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer: HaRe, and its API. In J. Boyland and G. Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005.

[9] M. Lopes Cornélio. *Refactorings as Formal Refinements.* PhD thesis, Centro do Informática, Universidade Federal de Permambuco, 2006.

[10] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[11] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press, 2004.

[12] PacSoft. Programatica: Integrating programming, properties and validation. www.cse.ogi.edu/PacSoft/projects/programatica/, 2005.

[13] C. Ryder and S. Thompson. Porting HaRe to the GHC API. Technical Report 8-05, Computing Laboratory, University of Kent, Canterbury, Kent, UK, October 2005.

[14] Simon Peyton Jones. Haskell 98 Language and Libraries: The Revised Report. http://www.haskell.org/onlinereport, 2002.

[15] S. Thompson. Refactoring Functional Programs. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 331–357. Springer Verlag, September 2005.

[16] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[17] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.