# Towards Type Based Refactorings

Christopher Brown
Computing Laboratory, University of Kent
cmb21@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 correctly. Preserving behaviour guarantees that refactoring does not introduce (or remove) any bugs.

## 1. INTRODUCTION

Refactoring [3] is the process of changing the structure of a computer program without altering its behaviour. A structural change can potentially make a function definition simpler, by removing duplicate code, say, or can be a preparatory step for an upgrade or extension of a system. The Haskell refactorer HaRe [2] is a refactoring tool for Haskell, developed at the University of Kent by Huiqing Li, Claus Reinke and Simon Thompson. HaRe currently works in (x)Emacs and Vim and supports a wide range of refactorings, such as renaming a definition, removing or adding arguments to a function, and converting a concrete data type into an abstract data type. HaRe uses the Programatica [4] front-end to do all the parsing and language manipulation. Haskell is used as the language to implement the refactorings and also as the language to refactor.

Until now, refactorings implemented for HaRe did not need type information and can rely on just the information supplied in the abstract syntax tree (AST). Haskell being a strongly typed language means that there are a number of refactorings and transformations that require the use of type information.

In this paper I describe some new refactorings for HaRe that use type information. One of these new refactorings also uses program slicing. Type checking is provided by the GHC-API [1]; the motivation for this is also discussed.

## 2. ADDING A CONSTRUCTOR

Adding a constructor to a data type is an interesting problem in Haskell. Whenever a new program is being developed, programs start small and incrementally get more complex, each time adding more information and functionality. When a programmer adds a new constructor to a data type, they also often want to pattern match over the data type, this can be done automatically by a refactoring. This transformation has the potential of altering the behaviour of all definitions that pattern match over the data type. Figure 1 shows a program with a data type named `Data` and a simple function definition, `f` that uses pattern matching over

```
data Data = AA | BB

f :: Data -> Int
f AA = 42
f BB = 0
```

**Figure 1: A Simple Program**

```
data Data AA | BB | NewCon Int

addedNewCon = error "added NewCon Int to Data"

f :: Data -> Int
f AA = 42
f BB = 0
f (NewCon a) = addedNewCon
```

**Figure 2: Constructor `NewCon` has been added to `Data`**

the data type. Figure 2 shows a similar program; this time a new constructor named `NewCon` (with an `Int` parameter) has been added to the data type `Data`, a new clause for the definition of `f` has also been added to cope with the possibility of `f` being called with the new constructor `NewCon` as a parameter.

The major problem with adding a constructor to a data type is not the trivial part of adding the constructor, it is constructing new patterns for definitions that reference the data type. In some cases type information is not needed to find the clauses that reference the data type because a constructor is directly referenced in the pattern (as in Figure 2). Type information becomes essential when a constructor of the data type in question is not referenced in the pattern match, or is not referenced in a nested pattern. Consider the identity function over the data type `Data`:

```
f x = g x

g AA = AA
g BB = BB
```

```
f CC = ?
f x = g x

g AA = AA
g BB = BB
g CC = ?
```

The refactoring needs to create new patterns for all definitions which work over the type `Data`. This can be done by

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

**Figure 3: 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 4: Splitting the functionality of a tuple into separate definitions**

```
f x = result x
      where
        result x = res
        res = (42,43)
```

**Figure 5: A definition that returns a tuple by calling a local definition**

using the type checker to check whether the arguments to a definition are of the type in question. If any of the arguments are of the type in question, then the refactoring can proceed with creating new pattern matching for the arguments of type `Data`.

# 3. PROGRAM SLICING

Another refactoring for HaRe which has been implemented using the type-checker was a static, backwards program slicer [5] . The program slicer can work in two distinct modes: it can compute a program slice for a particular expression of interest, or it can create new definitions for functions that return tuples, creating new definitions to encapsulate the functionality of each tuple element.

Consider the program in Figure 3. 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 4. The converse could also be achieved to some extent, merging the definitions `f1` and `f2` back into the definition `f`, but this is currently flagged for future work.

It is common practice for Haskell programmers to write a definition that returns multiple computations (Figure 5). It is also common practice to change one of the computations that the function returns. Having refactorings that can split a function into separate definitions, and also merging definitions together can make changing the functionality, or optimising these definitions much easier.

```
f1 x = result x
      where
        result x = res
        res = 42
f2 x = result x
      where
        result x = res
        res = 43
```

**Figure 6: Splitting the functionality of a definition that does not explicitly return a tuple**

Figure 5 shows an example of where the program slicer needs to use type information. Looking at the right hand side of `f` it is impossible to infer the type of `result x` without looking at the type of the definitions `result` and `res`. In cases like this where the right hand side of the definition is not an explicit tuple, the program slicer calls the type checker to find the types of the definitions needed. The slicer proceeds if the definition returns a tuple, or is a call to a local definition returning a tuple. Figure 6 shows the result of the program slicer over the definition `f`.

# 4. USING TYPE INFORMATION

In order to create refactorings that use type based information we make 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 a program. The type decorated AST also contains other information to help the type checker deal with overloading. The type checker does this by placing extra parameters to functions that use overloading therefore changing the structure of the AST, so that it no longer represents the source program.

However, the Programatica type checker was slow to use, so instead, HaRe now utilises the type checker from the GHC-API, simply calling it with an arbitrary expression to gather type information when needed.

I expect the work on using type information in refactorings to develop further. Firstly to create new refactorings that merge and optimise definitions. There are also a number of new refactorings that can be implemented such as adding or removing class instances, for example.

# 5. REFERENCES


[1] 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.

[2] 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 LDTA workshop 2005*, April 2005.

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

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

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