An Expression Processor:  
A Case Study in Refactoring Haskell Programs

Christopher Brown¹, Huiqing Li², and Simon Thompson²

¹ School of Computer Science, University of St. Andrews, UK  
chrisb@cs.st-andrews.ac.uk

² School of Computing, University of Kent, UK  
{H.Li,S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is the process of changing the structure of a program while preserving its behaviour in order to increase code quality, programming productivity and code reuse. With the advent of refactoring tools, refactoring can be performed semi-automatically, allowing refactorings to be performed (and undone) easily.

In this paper, we briefly describe a number of new refactorings for Haskell 98 programs implemented in the Haskell Refactorer, HaRe. In particular, a number of new structural and data-type refactorings are presented. We also implement a simple expression processor, clearly demonstrating how the refactorings and the HaRe tool can aid programmers in developing Haskell software. We conclude the paper with a discussion of the benefits of refactoring Haskell programs, together with their implementation and design limitations.

1 Introduction

Often programmers write a first version of a program without paying full attention to programming style or design principles [1]. 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 the design of a program without changing its functionality: often this is precisely what is needed in order to begin adapting or extending it.

The term ‘refactoring’ was first introduced by Opdyke in his PhD thesis in 1992 [2] and the concept goes at least as far back as the fold/unfold system proposed by Burstall and Darlington in 1977 [3], although, arguably, the fold/unfold system was more about algorithm change than structural changes. A key aspect of refactoring — illustrated by the ‘rename function’ operation — is that its effect is across a code base, rather than being focussed on a single definition: renaming a function will have an effect on all the modules that call that function, for instance.

The Haskell Refactorer, HaRe, is a product of the Refactoring Functional Programs project at the University of Kent [4] [5] by Li, Reinke, Thompson and Brown. HaRe provides refactorings for programs written in the full Haskell 98
standard language \cite{6}, and is integrated with the two most popular development environments for Haskell programs \cite{7}, namely Vim \cite{8} and (X)Emacs \cite{9}. HaRe refactorings can be applied to both single- and multi-module projects.

HaRe is itself implemented in Haskell, and is built upon the Programatica \cite{10} compiler front-end, and the Strafunski \cite{11} library for generic tree traversal. The HaRe programmers’ application programming 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 in the implementation of refactorings.

In this paper, we describe briefly a number of new refactorings for HaRe and demonstrate their use by applying them to an expression processing example. Using Haskell as the implementation language allows us to explore the usability of Haskell for implementing transformation and analysis tools.

We are also able to reflect on how refactoring functional programs – and in particular programs in Haskell – is different from refactoring within the OO paradigm. Pure functional languages such as Haskell make some refactorings substantially more straightforward: consider the example in which a function definition is generalised by selecting a sub-expression to pass as an argument, as in the transformation of the following program on selection of the sub-expression 1 within the definition of \texttt{addOne},

\begin{verbatim}
addOne [] = []
addOne (x:xs) = x+1 : addOne xs

fun xs = sum (addOne xs)
\end{verbatim}

where for good measure we also rename the function appropriately:

\begin{verbatim}
addNum n [] = []
addNum n (x:xs) = x+n : addNum n xs

fun xs = sum (addNum 1 xs)
\end{verbatim}

We note three aspects of this transformation.

- In performing a generalisation over an arbitrary sub-expression we can be sure that the expression has no side-effects, and so it can be passed as an argument without changing the order in which these effects take place.
- Because Haskell is evaluated lazily, we know that the argument will only be evaluated if it is used, and so we will not change the strictness of the function by generalising in this way; this would not be the case in a strict language.
- Finally, if we choose as a sub-expression something of functional type then because functions are ‘first-class citizens’ in Haskell the generalisation can take place: the use of arbitrary closures in (e.g.) object-oriented languages would make this generalisation awkward or indeed impossible.

If we were to introduce side-effects in a measured way – as in Haskell monads or in Erlang’s communication primitives – it is possible to detect where side-effects may take place, and indeed to ‘wrap’ the effects in a function closure when generalising, if that is required.