An Expression Processor: A Case Study in Refactoring Haskell Programs

Christopher Brown[1], Huiqing Li[2] and Simon Thompson[2]

[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. This behaviour preservation is crucial so that refactorings do not introduce any bugs. Refactoring is aimed at increasing code quality, programming 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, allowing refactorings to be performed (and undone) easily.
In this paper, we briefly describe a number of refactorings implemented in the Haskell Refactorer, HaRe. In addition to this, we also implement a simple expression processor to demonstrate how some of the refactorings implemented in HaRe can be used to aid programmers in developing Haskell software.

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 [5] [6] by Li, Reinke, Thompson and Brown. HaRe provides refactorings for programs written in the full Haskell 98

standard language [7], and is integrated with the two most popular development environments for Haskell programs [8], namely Vim [9] and (X)Emacs [10]. HaRe refactorings can be applied to both single- and multi-module projects.

HaRe is itself implemented in Haskell, and is built upon the Programatica [11] compiler front-end, and the Strafunski [12] 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 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 the exploration of the usability of Haskell for implementing transformation and analysis tools. The particular contributions presented of this paper are to cover:

- **Structural and Data-Type Refactorings**. The design and implementation of a set of structural and data-type refactorings. These refactorings are introduced in Section 2.
- **Refactoring Case Study**. A case study for refactoring Haskell programs. In particular we apply the refactorings described in this paper to an expression processing example. The example is used to demonstrate the capacity of the refactorings from this paper in a simple, but still useful, context. This case study is presented in Section 3.

We conclude the paper with a discussion of future work.

2 Structural and Data-Type Refactorings

This section describes some new structural and data-type refactorings that have been defined and implemented in HaRe. In this paper we chose to select the refactorings that would appear most useful to the Haskell programmer. The refactorings presented here follow on from the refactoring work by Li [13], and use the refactoring catalogue [14] maintained by Thompson as a basis. In particular, the following refactorings are described in this section: folding (Section 2.1); merging (Section 2.2); adding a constructor (Section 2.3); removing a constructor (Section 2.4); adding and removing a data type field (Section 2.5); and introducing pattern matching (Section 2.6).

We note that the refactorings are only very briefly described here. For a much more detailed overview of the transformation rules and side conditions for each refactoring described in this section, we refer the reader to Brown's PhD thesis [15].

2.1 Folding

Folding replaces instances of the right-hand-side of a definition by the corresponding left-hand-side. This refactoring is designed to be the complement of

unfolding which is described in Li's PhD thesis [13]. Folding can be used to eliminate some duplicate expressions within a program; it 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. This is achieved by first extracting the common definition using the *introduce new definition* refactoring [13], and then folding against this newly introduced definition.

Example An example of folding an instance of the right hand side of a definition, `table`, is shown in Figure 1. In the figure, two definitions are given:

Before:	After:
`showAll = (concat . format) . (map show)`	`showAll = table . map show`
`table  = concat . format`	`table   = concat . format`

Fig. 1. Before and after simple fold

`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`, except those that appear on the right-hand-side of `table`) are replaced with a call to `table`. The top row of the example shows that the sub-expression, `(concat . format)` has been replaced with a call to `table`, passing in `(map show)` as an argument; this therefore eliminates some duplicated code within the program.

2.2 Merging

Merging takes a number of selected definitions and creates a new, *generative*, definition that returns a tuple. Each component of the tuple returned by the merged definition encapsulates the behaviour of the selected entities. The merged definition is generative in the sense that it is recursive, and removes duplicate parts of the function by introducing code sharing. Merging is the inverse of splitting, as defined in [15].

Merging is actually known as *tupling* in the field of program transformation, and was originally proposed by Pettorossi [16], as a strategy for composing efficient computations by avoiding repeated evaluations of recursive functions.

Example An example of merging the functions `splitAt_1` and `splitAt_2` is shown, from left to right, in Figure 2. In order to perform the merge, the user must first select each function `splitAt_1` and `splitAt_2` in turn and add them to a merging *cache*, so that HaRe can perform the refactoring over the selected entities. The newly introduced definition, `splitAt`, uses only one list traversal, rather than a separate traversal for each of `splitAt_1` and `splitAt_2`.

Before:

```
splitAt_1 :: Int -> [a] -> [a]
splitAt_1 0 _ = []
splitAt_1 _ []= []
splitAt_1 n (x:xs)
   = x : splitAt_1 (n-1) xs

splitAt_2 :: Int -> [a] -> [a]
splitAt_2 0 xs = xs
splitAt_2 _ [] = []
splitAt_2 n (x:xs)
   = splitAt_2 (n-1) xs
```

After:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs) = (x:ys,zs)
     where
      (ys,zs) = splitAt (n-1) xs
```

Fig. 2. Merging a pair of definitions

2.3 Adding a Constructor to a data type

Adding a constructor to a defined data type. The introduced constructor is added immediately after a selected constructor definition in a data type. New pattern matching is introduced for all functions defined over the modified data type.

Example An example of adding a constructor `Var` to a data type `Expr` is shown in Figure 3. 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`. This is done by HaRe prompting the user for the constructor name and the types of its fields when the refactoring is selected from the menu. The function `eval` is updated automatically to include pattern matching for the newly added 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
```

Fig. 3. Adding a constructor

2.4 Removing a Constructor from a Data Type

Removing a constructor is defined as the inverse of adding a constructor. Rather than being a refactoring, removing a constructor is instead defined as a *destructive transformation* in the sense that it eliminates equations from the program space; this therefore may change the behaviour. Removing a constructor allows a constructor to be identified and all clauses that involve pattern matching over the constructor are commented out. All occurrences of the constructor in an expression are replaced with calls to `error`.

example An example of removing a constructor `Var` from a data type `Expr` is defined in Figure 3 read from right to left. `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`. The equation `eval (Var a) = addedVar` is also commented out, although this is not shown in the figure.

2.5 Adding or Removing a Field to or From a Constructor

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

Removing a field is defined as the inverse of adding a field. All references to the removed field in pattern matches are commented out of the program. Removing a field is a destructive transformation rather than a refactoring, as it changes behaviour.

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
```

Fig. 4. Adding and removing a field

Example Figure 4, read from left to right, 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 type signatures which involve the type `Data1` in the program.

Conversely, Figure 4, read from right to left, 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`.

2.6 Introduce Pattern Matching Over an Argument Position

Introduction of 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.

Before:

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

After:

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

Fig. 5. Introducing pattern matches for a pattern

Example An example of introducing pattern matches is given in Figure 5 from left to right. In the example, the new pattern matches are added to the definition of `f` and the introduced patterns for `x` are placed within an as pattern. 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 binding conflicts can occur. In the example, the pattern variables `y` and `ys` are introduced.

3 Refactoring an Expression Processor

In this section we present a simple example illustrating how the majority of the refactorings described in this paper could be used. In 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. In addition to the previously mentioned techniques, we also make use of the following refactorings from Li's thesis [13]: *renaming*; *generalising*; *introducing a new definition*; and *adding an argument to a definition.*

The example starts with the very basics of implementing a language, parser and evaluator. The code for this is shown in below; the grammar for the language

```
1   data Expr = Literal Int | Plus Expr Expr
2               deriving Show
3
4   parseExpr :: String -> (Expr, String)
5   parseExpr (' ':xs) = parseExpr xs
6   parseExpr ('+':xs) = (Plus parse1 parse2, rest2)
7                           where
8                             (parse1, rest1) = parseExpr xs
9                             (parse2, rest2) = parseExpr rest1
10  parseExpr (x:xs)
11     | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
12              where
13                lit = parseInt xs
14                parseInt :: String -> String
15                parseInt [] = []
16                parseInt (x:xs) | isNumber x = x : parseInt xs
17                                | otherwise  = []
18  parseExpr xs = error "Parse Error!"
19
20  eval :: Expr -> Int
21  eval (Literal x) = x
22  eval (Plus x y) = (eval x) + (eval y)
```

Fig. 6. The basic language and parser

is described in the `data` type on Line 1 in Figure 6. So far, the language only has the capacity to handle integer 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. 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 unsigned (positive) integers. For the purpose of this example the expressions are given in a prefix format. The complete implementation, for each stage of the case study, can be found at [17].

3.1 Stage 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 2.3). The refactoring asks us for the name of the constructor and any arguments. We enter `Mul Expr Expr` and select the constructor `Plus`. The refactoring always adds the new constructor immediately after the highlighted constructor. In this case the refactoring adds the new constructor to the end of the definition of `Expr` and also generates additional pattern matching clauses to `eval` (we use italics to show code introduced by the refactorer):

```
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 has also inserted a call to the (automatically created) definition of `addedMul` which is easily replaced with actual functionality in the succeeding steps.

3.2 Stage 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 2.3). 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`. This operation of removing constructors and introducing a new, generalised, type, may be implemented as refactoring, and is known as *introduce layered data type* in the catalogue of refactorings, maintained by Thompson [14].

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 2.6 on Page 6) 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.

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 call to `error` on the right hand side of `parseExpr` for the `'+'` case is then replaced with `Bin Plus parse1 parse2`. The next step is also 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)
```

Multiplication is then introduced in the parser, by copying the `'+'` case into a `'*'` case, and substituting `Plus` for `Mul` on the right hand side.

3.3 Stage 3: Generalisation

The next step is to do some *generalisation* and *folding*. As can be seen from Figure 7, 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 7 - 10 from Figure 7. We enter `parseBin` as the name for the new expression, and HaRe introduces the following code:

```
...
parseExpr ('+':xs) = parseBin xs
...
```

```
1   parseExpr :: String -> (Expr, String)
2   parseExpr (' ':xs) = parseExpr xs
3   parseExpr ('*':xs) = (Bin Mul parse1 parse2, rest2)
4                        where
5                          (parse1, rest1) = parseExpr xs
6                          (parse2, rest2) = parseExpr rest1
7   parseExpr ('+':xs) = (Bin Plus parse1 parse2, rest2)
8                        where
9                          (parse1, rest1) = parseExpr xs
10                         (parse2, rest2) = parseExpr rest1
11  parseExpr (x:xs)
12    | isNumber x = (Literal (read (x:lit)::Int), drop (length lit) xs)
13                where
14                  lit = parseInt xs
15                  parseInt :: String -> String
16                  parseInt [] = []
17                  parseInt (x:xs) | isNumber x = x : parseInt xs
18                                  | otherwise  = []
19  parseExpr xs = error "Parse Error!"
```

Fig. 7. The parser implementation with plus and multiplication

```
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 2.1) the equation `parseExpr` defined in Figure 7 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)
  | isNumber x = ...
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 simple: there is now a separate evaluator for binary operators (defined in Section 3.2) as well as a separate parser for binary operators; this allows for the code to be easily maintained in future versions.

3.4 Stage 4: Introduce Variables

We now add variables to the language by defining the `let` expression. In order to do this, the `Let` and `Var` constructs need to be added to the language, taking a variable name to be a String. The parser is then extended to handle the new constructs, with the input `let x=4 in 1+x` giving the AST

```
Let "x" (Literal 4) (Bin Plus (Literal 1) (Var "x"))
```

Having variables in the language means that bindings of variables to values need to be stored in an environment, and that environment variable needs to be passed into the evaluator as an extra parameter: when a variable is evaluated `lookup` is used to find its value in the environment.

To perform this extension to the language, first we perform two *add constructor* refactorings to the definition of `Expr`, adding `LetExp String Expr Expr` and then `Variable String` as arguments to the refactoring. The refactorings introduce new pattern matches for `eval`, thus:

```
data Expr = ... | LetExp String Expr | Var String
  ...
addedLetExp = error "Added LetExp String Expr Expr to Expr"
addedVar = error "Added Var String to Expr"
  ...
eval :: Expr -> Int
  ...
eval (LetExp p_1 p_2 p_3) = addedLetExp
eval (Var p_1) = addedVar
```

The `Environment` type, `addEnv` and `lookup` functions are now defined (not part of the refactoring sequence). Finally the definition of `eval` needs to be modified to take a new parameter, namely the `Environment`. This can be added using an "add argument" refactoring, but the definition needs then to be edited by hand to thread the environment through the computation, giving

```
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 p_1 p_2 p_3) = eval (addEnv p_1 p_2 env) p_3
eval env (Var p_1) = lookup p_1 env
```

3.5 Stage 5: 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. Initially `prettyPrint` is defined with the equation `prettyPrint x = error "Unable to pretty print!"`. We choose the *introduce pattern matching* from HaRe, which produces the following:

```
1   eval :: Environment -> Expr -> (String, Int)
2   eval env (Literal x) = (show x, x)
3   eval env (Bin op e1 e2) = ((fst (eval_op op)) ++ " "
4                              ++ (fst $ eval env e1) ++ " "
5                              ++ (fst $ eval env e2),
6                              (snd $ eval_op op) (snd $ eval env e1)
7                              (snd $ eval env e2))
8                   where
9                    eval_op :: (Num a) => Bin_Op -> (String, (a -> a -> a))
10                   eval_op p_1@(Mul) = ("*", (*))
11                   eval_op p_1@(Plus) = ("+",(+))
12                   eval_op _ = error "Undefined Operation"
13  eval env (LetExp n e e_2) = ("let " ++ n ++ " = " ++ (fst $ eval env e)
14                               ++ " in " ++ (fst $ eval env e_2),
15                               snd $ eval (addEnv n e env) e_2)
16  eval env (Var n) = (n, snd $ eval env (lookUp n env))
```

Fig. 8. The evaluator implementation with the generality of binary operators expressed

```
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 procedure 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 to 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 2.2). We also move the definitions of `eval_op` and `prettyPrintBinOp` to a `where` clause of the newly merged `eval` function.

4 Related Work

Program transformation for functional programs has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [18]. Other work in program transformation for functional languages is described by Hudak in his survey [19]. For an extensive survey of refactoring tools and techniques, Mens produced a refactoring survey in 2004 detailing the most common refactoring tools and practices [20].

The University of Kent and Eötvös Loránd University are now in the process of building refactoring tools for Erlang programs [21]. 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 [22] 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 fold/unfold system of Burstall and Darlington [3] 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; folding; instantiation; abstraction; definition and laws. 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. Indeed, the fold refactorings implemented for HaRe also suffer from the same termination problems.

5 Conclusions and Future Work

This paper has explored transforming (in the sense of both general program transformation, and refactoring) and analysing the functional programming language Haskell. A particular limitation lies with designing refactorings in their full generality instead of a large set of smaller, simpler refactorings, as it is not possible to always find a single most general transformation of an envisaged refactoring. 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 in this paper, are difficult to describe in terms of both the conditions and the transformation rules.

The approach that has been taken for this paper 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.

The work presented in this paper 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 [14] that are still awaiting implementation.
- 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.
- We hope to extend HaRe 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.
- Finally, we wish to port HaRe to GHC Haskell —the *de facto* standard of Haskell— and use the GHC API instead of Programatica for implementing refactorings.

The authors would like to thank Dave Harrison for his editorial advice, and the anonymous reviewers for their comments. We would also like to acknowledge EPSRC for supporting the original development of HaRe.

References

1. Brooks, F.P.: The mythical man-month: After 20 years. IEEE Software **12**(5) (1995) 57–60

2. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, Champaign, IL, USA (1992)
3. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. J. ACM **24**(1) (1977) 44–67
4. Holland, J.H.: Adaptation in natural and artificial systems. MIT Press, Cambridge, MA, USA (1992)
5. Li, H., Thompson, S., Reinke, C.: 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, http://www.sciencedirect.com/science/journal/15710661.
6. Li, H., Reinke, C., Thompson, S.: Tool Support for Refactoring Functional Programs. In: ACM SIGPLAN 2003 Haskell Workshop, Association for Computing Machinery (August 2003) 27–38
7. Peyton Jones, S., Hammond, K.: Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press (December 2003)
8. Refactor-fp Group, T.: The Haskell Editing Survey. `http://www.cs.kent.ac.uk/projects/refactor-fp/surveys/haskell-editors-July-2002.txt` (2004)
9. Oualine, S.: Vim (Vi Improved). Sams (April 2001)
10. Cameron, D., Elliott, J., Loy, M.: Learning GNU Emacs. O'Reilly (December 2004)
11. Hallgren, T.: Haskell Tools from the Programatica Project. In: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2003) 103–106
12. Lämmel, R., Visser, J.: A Strafunski Application Letter. In: Proc. of Practical Aspects of Declarative Programming (PADL'03). Volume 2562 of LNCS., Springer-Verlag (January 2003) 357–375
13. Li, H.: Refactoring Haskell Programs. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK (September 2006)
14. Refactor-fp Group, T.: Refactoring Functional Programs. `http://www.cs.kent.ac.uk/projects/refactor-fp` (2008)
15. Brown, C.: Tool Support for Refactoring Haskell Programs. `http://www.cs.kent.ac.uk/projects/refactor-fp/publications/ChrisThesis.pdf` (September 2008)
16. Pettorossi, A.: A Powerful Strategy for Deriving Efficient Programs by Transformation. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming, New York, NY, USA, ACM (1984) 273–281
17. Brown, C., Thompson, S.: Expression processor example code. `http://www.cs.st-and.ac.uk/~chrisb/tfp2010/` (2010)
18. Partsch, H., Steinbruggen, R.: Program Transformation Systems. ACM Comput. Surv. **15**(3) (1983) 199–236
19. Hudak, P.: Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Survey **21**(3) (1989) 359–411
20. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2) (2004) 126–139
21. Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., Víg, A.: Use cases for refactoring in erlang. In: CEFP. Volume 5161 of Lecture Notes in Computer Science., Springer (2007) 250–285
22. Sun Microsystems: MySQL. `http://www.mysql.com/` (2008)