A Catalogue of Functional Refactorings
Version 1

Simon Thompson, Claus Reinke
Computing Laboratory, University of Kent

1 August 2001

Abstract

This document is the first draft of a catalogue of refactorings for functional programs. Most are applicable to a variety of modern functional programming languages – with the example code begin written in Haskell – but some relate specifically to Haskell and are marked as such.

What is Refactoring?

Refactoring is 'improving the design of existing code' and as such, it has been practised as long as programs have been written. It was first identified as an activity in its own right within the object-oriented programming community [3, 4] (`http://www.refactoring.com`).

When does refactoring arise? To take an example, we might first program a system using an algebraic data type, and then decide to change the way that the data are represented. How should we proceed with this? One option is to modify the `data` type directly, that is to achieve the modification in a single step: this will require us to make substantial modifications to a program's functionality *and* structure simultaneously.

On the other hand, we might do the same in two stages. First we could transform the algebraic data type into an abstract data type (ADT), and only after this *refactoring* is done, would we modify the definition of the ADT. This two-stage transformation aims to separate the structural changes (from algebraic to abstract data type) from the changes in functionality. It also makes the program more amenable to further change, as ADT representations can be modified with no cost to the client.

We refactor in other situations too. We might program in an *exploratory* way: first establishing the functionality we seek, and then refactoring into a more elegant form. We suspect that functional languages are particularly suited to this form of programming, because their clean semantic basis makes wholesale transformations more feasible than for a language in the C family, say.

Finally, we might refactor someone else's program to make it more *readable*; some of the refactorings given in this note were inspired by trying to fully understand some non-trivial student assessments written in Haskell.

The Nature of Refactoring

One of the simplest refactorings – Refactoring 1 in this catalogue – is to rename a function to reflect its use. We have already discussed the rather more complex Refactoring 9 from an algebraic to an abstract data type. These examples share two important characteristics of refactorings.

Diffuse Their effect is diffuse, in that they require changes throughout a module and indeed throughout a system of modules. A change of function name needs to be effected at each function call; a change from a `data` type to an ADT will require changes to all functions that directly manipulate the data by pattern matching, for instance.

Bidirectional A change from a general name (e.g. `f`) to a more specific one (e.g. `findMaxVolume`) might later be followed by a change to a more general name (e.g. `findMax`).

We have discussed the change from an algebraic data type to an ADT, but in other situations it is perfectly reasonable to change an ADT into an algebraic type. One motivation might be to use pattern matching, another to use the `deriving` facilities over `data` in Haskell.

Supporting Refactoring

We have seen that refactorings have a bureaucratic aspect: changes have to be made at all sites that a function is called, for example. With current technology we would use a text editor to assist in making the changes, and rely on a type checker to catch any errors introduced in the refactoring. OO refactorers underline the importance of continual (re)testing of code to ensure correctness [1].

For a functional programming language one could use reasoning to establish the correctness of many classes of refactorings. Moreover, it is entirely feasible to support these refactorings in a variety of *tools* of increasing levels of sophistication; the experience of the OO community [2] in this respect is broadly positive. A tool could allow users to do and undo refactorings of various sorts; it could also check the applicability of certain transformations, such as renaming or lifting. More detailed considerations of tool design are to be found in [5].

The catalogue

The body of this paper is a list of refactorings applicable to functional programs written in the ML/Haskell/Miranda school of strongly-typed functional language.

Refactoring 1: Renaming

```
f x y = ...
```

$\Longleftarrow$

The name is general enough to be re-used in many different circumstances; it may be that the name is too specific for reuse, or it is mis-named.

```
findGreaterVolume x y = ...
```

$\Longrightarrow$

The name is chosen to reflect the purpose of the function in the module (`F`, say) that it is used.

This requires all calls of `f` to be renamed to `findGreaterVolume`; this will certainly have an effect in the module `F`. By default, it can also affect any module importing `F`.
It can have an effect beyond those modules explicitly importing `F` depending upon the visibility given to `f` by modules that import `F`.

Refactoring 2: Lifting / Demoting

```
f x y = ... h ...
        where
        h = ...
```

$\Longleftarrow$

The effect of defining `h` locally is to clear up the namespace (of the module containing it, `F`, say, and of the whole system). In many situations a top-level function (`f` here) uses an auxiliary function whose only use is within `f`. An example is where `fAux` takes an extra parameter, and `f` is a call to `fAux` with an initial value of the parameter. A concrete example is given by a search function which takes a list of already-visited nodes as arguments; at the top level this will be called with an empty list.
Another justificication is in creating *circular* data structures: a `where`-defined `h` such as `h = 1:h` will create a circular representation of the infinite list of ones.

```
f x y = ... (h x y) ...

h x y = ...
```

$\Longrightarrow$

Lifting `h` to the top-level makes it accessible to the other functions in the module `F`, say, and to all modules importing `F`.

Changing the scope of a definition of `h` in either direction has potential effects. Making `h` local will be a problem if it is used by any function other than `f`. Lifting it to the top level can potentially cause name clashes; all calls to `h` within `f` need to be modified to pass the extra arguments: in this case `x` and `y`.

Refactoring 3: Naming a type

```
f :: Int -> Char
g :: Int -> Int
```

$\Longleftarrow$

Reuse supported.

```
type Length = Int
f :: Length -> Char
g :: Int -> Length
```

$\Longrightarrow$

Clearer specification of the purpose of `f`,`g`. (Morally) can only apply to lengths, but in fact a synonym is transparent, so can apply in exactly the same circumstances as the left hand version.

The intention is delivered by Refactoring 4.

Refactoring 4: Opaque Type Naming

```
f :: Int -> Char
g :: Int -> Int
```

```
data Length
  = Length { length::Int }
f’ :: Length -> Char
g’ :: Int -> Length

f’ = f . length
g’ = Length . g
```

$\Longleftarrow$
reuse supported.

$\Longrightarrow$
Clearer specification of the purpose of f,g. Can only apply to lengths.

The changes to types `f :: Length -> Char` and `g :: Int -> Length` require us to modify

- the calls to `f`, to replace an integer `e` by `Length e`;
- and the callers of g, to replace `F[g x]` by `F[length(g x)]`.

This can be done without labelled fields, but will in that case need to define an auxiliary selector

```
length (Length b) = b
```

or to use `let` expressions throughout the code. Taking `length` as the field name shows a potential clash of names: `length` is also defined in the standard prelude.
See also Refactoring 3.

Refactoring 5: Explicit or implicit arguments

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

```
f = g . h
```

$\Longleftarrow$
Definitions which use variables are often easier to read.

$\Longrightarrow$
An explicit use of a higher-order operation, in this case ‘.’, can pave the way for further transformations.

There are no further code changes required by this refactoring.

Refactoring 6: Generalise / Inline

```
f x ... = ... e ...
```

```
f’ g x ... = ... (g x) ...

g x = e
```

$\Longleftarrow$
Might have `f’` with only a single application; could then transform to `f` and remove the definition of `f’` completely.

$\Longrightarrow$
The particular behaviour `e` involves `x` as the only free variable. It is encapsulated in the function g.
The same transformation could be applied to an expression `e` with a number of free variables: all would go into the formal parameters of the newly defined function g.

This transforms the particular function `f` into the more general `f’`. Such a generalisation often results in the type becoming generalised also. For instance

```
add2s :: Int -> Int
add2s xs = [ 2+x | x<-xs ]
```

```
add2s’ :: (a -> b) -> [a] -> [b]
add2s’ add2 xs = [ add2 x | x<-xs ]
add2 x = 2+x
```

The example of `add2s’` is ready for renaming to a more general operation (e.g. `applyAll` or `map`).

Refactoring 7: Add/Remove Field Names in `data` Types

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

sumTree (Leaf n) = n
sumTree (Node t1 t2)
  = sumTree t1 + sumTree t2
```

$\Longleftarrow$
Simpler format; pattern matching used to select components of arguments as well as to select between the two different cases.

```
data Tree a = Leaf { leaf :: a } |
              Node { left, right :: Tree a }

sumTree t = case t of
              (Leaf _) -> leaf t
              (Node _ _) -> sumTree (left t) +
                            sumTree (right t)
```

$\Longrightarrow$
Halfway house to making the `data` type abstract.
It is still necessary to distinguish between different cases using pattern matching, either explicit (as here) or implict (by means of `isLeaf`, `isNode` functions which are themselves defined by pattern matching).

Field names have the additional advantage that the same name can be used within different constructors of the same data type, as in

```
data Vtree a = VLeaf { value :: a } |
               VNode { value :: a, left, right :: Tree a }
```

where the field `value` names the field of type `a` in both constructors of the `VTree` type.
See also Refactorings 8, 9.

Refactoring 8: Pattern Matching or Selectors and Discriminators

```
count (Add e1 e2) = 1 + count e1 + count e2
count (Mul e1 e2) = 1 + count e1 + count e2
count ...         = ...
```

$\Longleftarrow$
Pattern matching is the simplest approach for a small definition.

```
count e
  | isBinary e  = 1 + count (left e) + count (right e)
  | otherwise   = ...

isBinary (Add _ _) = True
isBinary (Mul _ _) = True
isBinary _         = False
```

$\Longrightarrow$
Similar cases – here the case of a binary operator – are treated as one.
If the fields (within `Add` and `Mul`) are named `left` and `right` then the selectors need not be defined; `isBinary` has to be defined by pattern matching.
This is closer to an abstract data type.

Related to Refactorings 7, 9.

Refactoring 9: Algebraic or Abstract Type?

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

flatten :: Tr a -> [a]

flatten (Leaf x) = [x]
flatten (Node s t)
 = flatten s ++
   flatten t
```

```
A type which exports the operations:
  isLeaf, isNode,
  leaf, left, right,
  mkLeaf, mkNode, ...

flatten :: Tr a -> [a]

flatten t
 | isleaf t = [leaf t]
 | isNode t
   = flatten (left t) ++
     flatten (right t)
```

$\Longleftarrow$
Pattern matching syntax is more direct …but can achieve a considerable amount with field names.
Can declare `Tr` as an instance of a class anywhere that it is used.
Can use the `deriving` facilities of Haskell to derive definitions of equality, a `show` function and so forth.

$\Longrightarrow$
Allows changes in the implementation type without affecting the client: e.g. might memoise values of a function within the representation type (itself another refactoring XREF?2).
Can only declare the type to be an instance of a class within the abstraction capsule, rather than at the point of use.
Allows an invariant to be preserved: suppose trees are to be kept ordered (search trees); this can be ensured if all the functions in the signature preserve it. If the carrier type is exposed, it becomes the user's responsibility to maintain the invariant, rather than the supplier of the type.

Such a refactoring has widespread effects. All examples of functions creating or accessing trees will need to be modified.
This is a good case for automatic support, in identifying definitions anf indeed in transforming pattern matching definitions into selector/discriminator definitions. This is made particularly easy if the fields have already been labelled, as one only needs to provide discriminators which can be named `isLeaf`, `isNode` etc. in a completely routine way.
See also Refactorings 7, 8.

Refactoring 10: Migrate functionality

```
A type which exports the operations:
  isLeaf, isNode,
  leaf, left, right,
  mkLeaf, mkNode, ...

depth :: Tr a -> Int

depth t
 | isleaf t = 1
 | isNode t
   = 1 +
     max (depth (left t))
         (depth (right t))
```

```
A type which exports the operations:
  isLeaf, isNode,
  leaf, left, right,
  mkLeaf, mkNode, ..., depth
```

$\Longleftarrow$
If the type is reimplemented, need to reimplement everything in the signature, including `depth`. The smaller the signature the better, therefore,

$\Longrightarrow$
Can modify the implementation to memoise values of `depth`, or to give a more efficient implementation using the concrete type.

This refactoring should have no effect on the surrounding code; it simply moves responsibility for the definition of `depth` from the client to the implementor of the type.
Related to Refactoring 9.

Refactoring 11: Algebraic or Existential type?

```
data Shape
  = Circle Float |
    Rect Float Float

area :: Shape -> Float
area (Circle f) = pi*r^2
area (Rect h w) = h*w

perim :: Shape -> Float
perim (Circle f) = 2*pi*r
perim (Rect h w) = 2*(h+w)
```

$\Longleftarrow$
Pattern matching is available. Adding a new sort of `Shape` requires modification of all functions which have `Shape` arguments.
Possible to deal with binary methods: impossible to deal with == on `Shape` as an existential type.

```
data Shape
  = forall a. Sh a => Shape a

class Sh a where
  area :: a -> Float
  perim :: a -> Float

data Circle = Circle Float

instance Sh Circle
  area (Circle f)  = pi*r^2
  perim (Circle f) = 2*pi*r

data Rect = Rect Float

instance Sh Rect
  area (Rect h w) = h*w
  perim (Rect h w) = 2*(h+w)
```

$\Longrightarrow$
Can add new sorts of `Shape` e.g. `Triangle` without modifying existing working code, since (unary) functions are distributed across the different `Sh` types.

This can be seen as the result of a sequence of simpler refactorings; see CITATATION for further details.

Refactoring 12: Replace Function by Constructor

```
data RegExp = Star RegExp |
       Then RegExp RegExp | ...

plus e = Then e (Star e)
```

$\Longleftarrow$
A function over `RegExp` only needs to be defined over `Star`, `Then`, ...; the case of `plus` is handled automatically by the functions over the `RegExp` type.
A function over `RegExp` is therefore more compact.

```
data Expr = Star RegExp |
            Plus RegExp |
       Then Expr RegExp | ...
```

$\Longrightarrow$
Can treat `Plus` differently, e.g.

```
literals (Plus e) = literals e
```

rather than the definition given by the ‘syntactic sugar’ of `plus` which expands thus

```
literals (plus e)
  = literals (Then e (Star e))
  = nub (literals e ++ literals (Star e))
  = nub (literals e ++ literals e)
  = ...
```

The disadvantage of this approach is that each function over `RegExp` needs to have a `Plus` clause defined.

The change will require modifications to all functions working over `RegExp`.
A more pertinent example here is a range of characters within a regular expression: one can expand `[a-z]` into `a|b|c|...|y|z` but it is much more efficient to treat it as a new constructor of regular expressions.

Refactoring 13: Layered `data` types

```
data Expr = Lit Float |
            Add Expr Expr |
            Mul Expr Expr |
            Sub Expr Expr

eval (Lit r) = r
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Sub e1 e2) = eval e1 - eval e2
```

$\Longleftarrow$

This approach is simpler: the different sorts of arithmetic expressions are all collected into a single data type. Its disadvantage is that the type does not reflect the common properties of the `Add`, `Mul` and `Sub` nodes, each of which has two `Expr` fields.

```
data Expr  = Lit Float |
             Bin BinOp Expr Expr
data BinOp = Add | Mul | Sub

eval (Lit r) = r
eval (Binop op e1 e2)
  = evalOp op (eval e1) (eval e2)

evalOp Add = (+)
evalOp Mul = (*)
evalOp Sub = (-)
```

$\Longrightarrow$

After the refactoring, the `Bin` node is a general binary node, with a field from `BinOP` indicating its sort. Operations which are common to `Bin` nodes can be written in a general form, and the pattern matching over the original `Expr` type can be reconstructed thus:

```
eval' (Bin Add e1 e2) = eval' e1 + eval' e2
```

This approach has the advantage that it is, in one way at least, more straightforward to modify. To add division to the expression type, it is a matter of adding to the enumerated type an extra possibility, `Div`, and adding a corresponding clause to the definition of `evalOp`.

The modification requires the rewriting of all definitions that either use or return an `Expr`.

Refactoring 14: Set comprehensions (Haskell-specific)

```
makeSet
[ f x y |
    x <- flatten xs,
    y <- flatten ys,
    p x y ]
```

$\Longleftarrow$

The notation looks more like

`{f x y | x<-xs, y<-ys, p x y}`

the monadic notation has a different connotation.

```
do x <- xs
   y <- ys
   guard (p x y)
   return (f x y)
```

$\Longrightarrow$

Doesn't require the abstraction to be broken by

`flatten :: Set a -> [a]`

Depending upon the particular implementation of sets, it might not be possible to define a `flatten` function.

Neither is ideal: one would like *set* comprehensions in the form `{f x y | x<-xs, y<-ys, p x y}`.

Refactoring 15: Enlarge the return type of a function

```
f :: S -> T
```

$\Longleftarrow$

This form is the most appropriate for a function which returns a single element of type `T`.

```
f :: S -> Maybe T
f :: S -> [T]
f :: S -> Either T T'
```

$\Longrightarrow$

This form is suitable prior to generalising or modifying the behaviour of `f` to accommodate

- the possibility of `f` returning no result (`Nothing`);
- the possibility of `f` returning a number of results of type `T`, as a list;
- the possibility that `f` would return a value either in `T` or in `T'`.

This modification is simple to achieve for the definition of `f`, which is pre-composed with `Just`, `(:[])` or `Left`. All call sites of the function will need to be modified; it would be advantageous if the built-in types had field names, as in

```
data Maybe a = Just { just::a } | Nothing
```

so that the call `F[f s]` could be replaced by `F[just (f s)]` etc.
See also Refactoring 4. This refactoring may well precede addition of error reporting or avoidance measures.

Refactoring 16: Reorganise Arguments

```
f :: S -> T -> U

g :: S -> T -> U
```

$\Longleftarrow$

The curried form of functions gives a cleaner notation for function application, and, more importantly, allows functions to be partially applied.
A function can only be partially applied to arguments from the left of the argument list, and so the order of arguments is significant. Its first two arguments can be swapped by applying the `flip` combinator, but to achieve an arbitrary permutation of its arguments, a function needs in general to be redefined.

```
f :: T -> S -> U

g :: (S,T) -> U
```

$\Longrightarrow$

The redefined order of arguments allows partial application to the argument of type `T`.
The uncurried form of the function is particularly useful for composition with a function that returns a pair of elements.

These require modifications at all sites where `f` and `g` are called. The changes required are entirely mechanical, and are a good first candidate for automation in a refactoring tool.
See also Refactoring 17

Refactoring 17: Add / Remove Arguments

```
h :: S -> U
```

```
h :: S -> T -> U
```

$\Longleftarrow$

If the result of the function does not depend on the second argument, then it can be removed. This situation would not occur deliberately, but might well be the result of other refactorings. input of type

$\Longrightarrow$

This refactoring appears to be pointless, but it can be made *prior* to modifying the definition of `h`; if done in this way, the structure of the program is modified before the substantive change in functionality is made *to the modified structure*.

It is possible to detect situations in which an argument to a function is not used; this could form a component of a refactoring tool.
See also Refactoring 16.
One example of adding an argument is as a value to be returned when an error condition holds. For instance

```
head :: [a] -> a

head (x:xs) = x
head []     = error "head of empty list"
```

```
headEr :: a -> [a] -> a

headEr y (x:xs) = x
headEr y []     = y
```

Refactoring 18: Monadic presentation

```
data Expr = Lit Float |
            Add Expr Expr |
            Mul Expr Expr |
            Sub Expr Expr

eval :: Expr -> Float

eval (Lit r) = r
eval (Add e1 e2) = eval e1 + eval e2
  ...
```

```
evalM :: Expr -> M Float

evalM (Lit r) = return r
evalM (Add e1 e2) = do v1 <- evalM e1
                       v2 <- evalM e2
                       return (v1+v2)
  ...
```

$\Longleftarrow$

This is the direct definition of evaluation. If one wants to add

- error handling,
- updateable variable within expressions, or
- other instrumentation

then it is necessary to modify the return type (and argument types) of the function.

$\Longrightarrow$

The monadic presentation here allows the modification of the monad `M` whilst preserving the top-level structure of the computation.

Modifications here are potentially wide-ranging. The type of the `eval` function suggests that a (single) result is returned for all expressions; this will *not* be the case for a general monad. It therefore becomes necessary for the part of the program using the results of `evalM` to be put into monadic form, unless there is an ‘escape’ function of type `M a -> a` available for the monad in question.

Refactoring 19: Iteration / Fixedpoint

```
it 0 f x = x
it n f x = it (n-1) f (f x)

solution = it e f v
```

```
fix f x
  | x==next    = x
  | otherwise  = fix f next
    where next = f x

solution = fix f v
```

$\Longleftarrow$
If the number of iterations supplied, `e`, is defined, then the `solution` is defined. The computation proceeds without equality tests, but needs a (tight) bound on the number of iterations required to be efficient. Making `it` strict in its third argument will make for space efficiency.

$\Longrightarrow$
Termination is only guaranteed if there is a fixed point in the sequence `x, f x, f (f x), ...`, but it will halt as soon as a fixed point is reached.

This only has an effect on the definition of the `solution` and not on any of the places that it is used.

Refactoring 20: Recursion / Fold operator

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

```
map f = foldr ((:).f) []
```

$\Longleftarrow$
This has the advantage of readability, since the recursive call (at `xs`) appears explicitly in the definition.

$\Longrightarrow$
The form of the recursion is evident from the definition, and indicates termination of the definition (if the arguments to `foldr` are of the appropriate kind).

This only has an effect on the definition (of `map` here) and not on any of the places that it is used.

Refactoring 21: Case switch flags

```
f 1 e ... = g e ...
f 2 e ... = h e ...
```

```
f' e ...
  | c1      = g e ...
  | c2      = h e ...
```

$\Longleftarrow$
The values `1`, `2`, and so forth are generated separately from `f`.

$\Longrightarrow$
The calculations of the first argument flag in `f` are folded into the function `f'` by means of the conditions `c1`, `c2` etc.

See also Refactoring 22.

Refactoring 22: Layered case switches

```
f (C1 ...) = ...
f (C2 ...) = ...
  ...
f (Cm ...) = ...
```

```
f t
  | splitCase t  = fSplit t
  | otherwise    = fNoSplit t
```

$\Longleftarrow$
Suppose that the `m` cases of pattern matching fit into two different modes, ‘splitting’ and ‘non-splitting’: this is not immediately apparent from this definition.

$\Longrightarrow$
The top-level structure of the algorithm is manifest in the top-level case split between splitting and not: separate functions handle the two cases.

See also Refactoring 21.

Conclusion

The list of refactorings presented here is plainly incomplete and unclassified. Our longer-term aims in this work are to compile, through case studies and other means, a more complete set of refactorings for functional languages; to classify the refactorings and, finally, to build tool support for the most useful refactorings. To be used by practising programmers the tool support would need to be integrated into the current programmer's toolset.

Bibliography

[1] Kent Beck. *eXtreme Programming eXplained: Embrace Change*. Addison-Wesley, 2000.

[2] John Brandt and Don Roberts. Refactory. `http://st-www.cs.uiuc.edu/users/brant/Refactory/`.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 2000.

[4] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] Simon Thompson and Claus Reinke. Refactoring Functional Programs. Grant application: available from the authors on request.