Refactoring Functional Programs

Simon Thompson

Computing Laboratory
University of Kent, UK
S.J.Thompson@kent.ac.uk

Abstract. Refactoring is the process of improving the design of existing programs without changing their functionality. These notes cover refactoring in functional languages, using Haskell as the medium, and introducing the HaRe tool for refactoring in Haskell.

1 Introduction

Refactoring [Fow00] is about improving the design of existing computer programs and systems; as such it is familiar to every programmer, software engineer and designer. Its key characteristic is the focus on structural change, strictly separated from changes in functionality. A structural change can make a program simpler, by removing duplicate code, say, or can be the preparatory step for an upgrade or extension of a system.

Program restructuring has a long history. As early as 1978 Robert Floyd in his Turing Award lecture [Flo79] encouraged programmers to reflect on and revise their programs as an integral part of their practice. Griswold's thesis on automated assistance for LISP program restructuring [GN93] introduced some of the ideas developed here and Opdyke's thesis [Opd92] examined refactoring in the context of object-oriented frameworks. Martin Fowler brought the field to prominence with his book on refactoring object-oriented programs [Fow00]. The refactoring browser, or 'refactory' [BR], for Smalltalk is notable among the first generation of OO tools; a number of Java tools are now widely available. The best known of these is the refactoring tool for Java in Eclipse [Ecl]. More comprehensive reviews of the refactoring literature are available at the web page for [Fow00] and at our web site.[1]

Refactorings are one sort of program transformation; they differ from other kinds of program transformation in a number of ways. Traditional transformations usually have a 'direction': they are applied to make a program more time or space efficient, say. On the other hand, refactorings are typically bi-directional: a refactoring to widen the scope of a local definition could equally well be applied in reverse to localise a global definition.

It is also characteristic of refactorings that they are 'diffuse' and 'bureaucratic': that is, their effect is not limited to a particular point in a program, and

[1] http://www.cs.kent.ac.uk/projects/refactor-fp/

they require care and precision in their execution. Consider the example of the simplest possible refactoring: renaming a component of a program. To effect this change requires not only the component definition to be changed, but also every *use* of the component must be similarly modified. This involves changing every file or module which might use the component, potentially tens or hundreds of modules. Moreover, it is vital not to change any components hidden in other parts of the system which happen to share the same name.

It is, of course, possible to do refactorings 'by hand', but this process is tedious and, more importantly, error-prone. Automated support for refactorings makes them safe and easy to perform, equally easy to undo, and also secure in their implementation. The *Refactoring Functional Programs*[2] [LRT03] project at the University of Kent is building the *HaRe* [HaR] system to support refactorings for Haskell programs.

HaRe is designed as a serious tool for use by practising programmers: HaRe supports the whole of Haskell 98; it is integrated into standard development environments and it preserves the 'look and feel' of refactored programs. HaRe is built using a number of existing libraries: Programatica [Hal03] on which to build the language-analysis components, and Strafunski [LV01] which gives general support for tree transformations.

These notes begin presenting overviews of design for functional programs and the HaRe system. The core of the paper is an exposition of the basics of refactoring: a detailed description of generalisation is presented as an example of a structural refactoring in Section 4, and the impact of modules on refactoring is examined in Section 5.

A number of data-oriented refactorings are given Section 6: principal among these is the transformation taking a concrete `data` type into an ADT, which is implemented in HaRe as composition of simpler refactorings. As well as providing a repertoire of built-in refactorings, HaRe provides an API by which other refactorings can be constructed; this is the subject of Section 7. The notes conclude with a discussion of conclusions and directions for the research.

I am very grateful indeed to my colleagues Huiqing Li and Claus Reinke, interns Nguyen Viet Chau and Jon Cowie, and research students Cyris Ryder and Chris Brown for their collaboration in the project. I would also like to thank the referees for their suggestions and corrections.

2 The elements of design

In designing an object-oriented system, it is taken for granted that design will precede programming. Designs will be written using a system like UML [UML] which is supported in tools such as Eclipse [Ecl]. Beginning programmers may well learn a visual design approach using systems like BlueJ [Blu]. Work on a similar methodology for functional programming is reported in [Rus00], but little other work exists. There may be a number of reasons for this.

[2] This work is supported by EPSRC under project grant GR/R75052.

– *Existing functional programs are of a scale which does not require design.* Many functional programs are small, but others, such as the Glasgow Haskell Compiler, are substantial.
– *Functional programs directly model the application domain, thus rendering design irrelevant.* Whilst functional languages provide a variety of powerful abstractions, it is difficult to argue that these provide all and only the abstractions needed to model the real world.
– *Functional programs are built as an evolving series of prototypes.*

If we accept the final reason, which appears to be the closest to existing practice, we are forced to ask how design emerges. A general principle is the move from the concrete to the abstract, and from the specific to the general. Specifically, for Haskell, we can use the following strategies:

Generalisation. A function is written with a specific purpose: it is generalised by making some of the particular behaviour into an argument.

Higher-order functions. This particular case of generalisation is characteristic of modern functional programming: specific behaviour is abstracted into a function, which becomes a parameter.

Commonality. Two parts of a program are identified as being identical or at least similar; they can be replaced by invocations of a single function (with appropriate parameters).

Data abstraction. Concrete, algebraic data types provide an excellent starting point, but are difficult to modify: a move to an abstract type gives the programmer flexibility to modify the implementation without modifying any client code.

Overloading. The introduction of a `class` and its `instances` allows set of names to be overloaded: programs thus become usable in a variety of contexts. This can make programs more readable, and also replace a number of similar definitions by a single, overloaded, one.

Monadification. This particular case of overloading allows explicit computational effects to become an implicit part of a system; once this transformation has taken place it is possible to modify the monad being used without changing the client code. A number of monads can be combined using monad transformers [LHJ95].

The HaRe tool supports many of these 'design abstractions'. Using a refactoring tool allows programmers to take a much more exploratory and speculative approach to design: large-scale refactorings can be accomplished in a single step, and equally importantly can be undone with the same effort. In this way Haskell programming and pedagogy can become very different from current practice.

3 The HaRe system

Refactoring for Haskell is supported by the HaRe tool [HaR] built at the University of Kent as a part of the project *Refactoring Functional Programs*. The

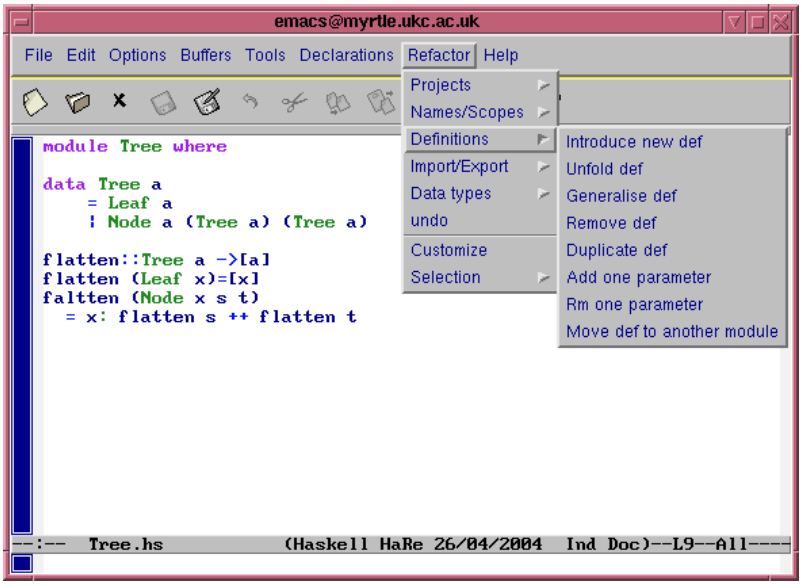

Fig. 1. HaRe: the *Refactor* menu

system was designed to be a tool useable by the working programmer, rather than a proof-of-concept prototype. This imposes three substantial constraints on the designer.

- It should support a full standard language – Haskell 98 in this case – rather than a convenient subset chosen for demonstration purposes.
- It should work within programmers' existing tools (Emacs and Vim) rather than be stand alone, allowing programmers to augment their existing practice with zero overhead.
- It is our experience that although layout is of syntactic significance in Haskell, different programmers adopt widely different styles of layout, and in most cases programmers would find it completely unacceptable to have had their code reformatted by a 'pretty printer' in the course of a refactoring.
  The system should therefore preserve the appearance of source code programs. In particular, it is crucial to preserve not only comments but also the particular layout style used by the programmer.

3.1 Using HaRe

HaRe supports a growing set of refactorings over Haskell; the details of many of these are presented in the sections that follow. The initial release of HaRe contained a number of 'structural', scope-related, single-module refactorings (October 2003); multiple-module versions of these refactorings were added in HaRe

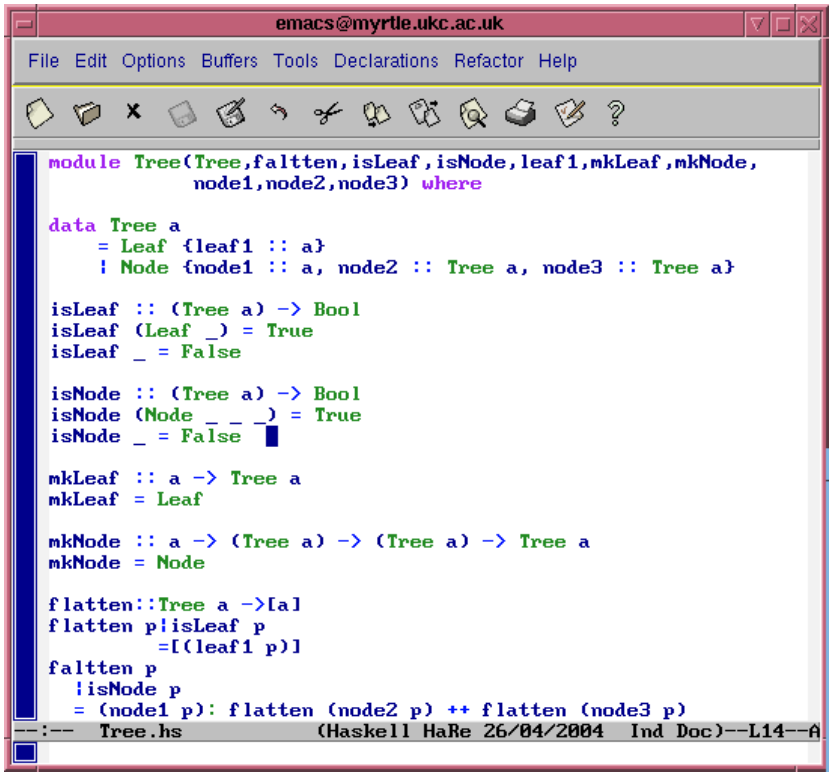

Fig. 2. HaRe: the result of '*From concrete to abstract data type*'.

0.2 (January 2004), and the first datatype-related refactorings added in HaRe 0.3 (November 2004). The third version restructures HaRe to expose an API for the system infrastructure used for implementing refactorings and other transformations in HaRe; this is addressed in more detail in Section 7.

HaRe, embedded in Emacs, is shown in Figures 1 and 2. A new *Refactor* menu has been added to the user interface: menu items group refactorings, and submenus identify the particular refactoring to be applied. Input is supplied by the cursor position, which can be used to indicate an identifier to be renamed, say, and from the keyboard, to give the replacement identifier, for instance. Figure 1 shows a program defining and using a concrete data type; Figure 2 shows the result of refactoring this to an abstract data type.

3.2 Implementation

HaRe is implemented in Haskell. It can be used as a stand-alone program, and is integrated with Emacs and Vim using their scripting languages. As is apparent from the example shown in Figures 1 and 2, HaRe is more than a *text* editor.

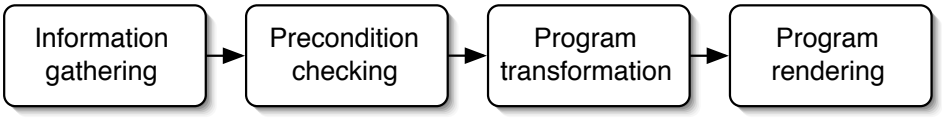

Fig. 3. The four stages of a refactoring

Implementing refactorings requires information about a number of aspects of the program:

Syntax. The subject of the refactoring (or program transformation) is the abstract syntax tree (AST) for the parsed program. To preserve comments and layout, information about comments and source code locations for all tokens is also necessary.

Static semantics. In the case of renaming a function `f` it is necessary to check that this binding of `f` does not capture any existing uses of `f`. The binding analysis provides this information.

Module analysis. In a multi-module project, analysis must include all modules. For example, renaming the function `f` must percolate through all modules of a project which import this binding of `f`.

Type system. If a function `g` is generalised (as in Section 4) then its type declaration will need to be adjusted accordingly.

It is therefore clear that we require the full functionality of a Haskell front-end in order to implement the refactorings completely and safely. In this project we have used the Programatica front end [Hal03], which supports all aspects of analysis of Haskell 98. The correct implementation of a refactoring consists of four parts, shown in Figure 3.

Information gathering and condition checking. The refactoring will only be performed if it preserves the semantics of the program; examples of some of the conditions are given above. Verifying these conditions requires information, such as the set of identifiers in scope at a particular point in the program, to be gathered from the AST by traversing it.

Transformation. Once the conditions are verified, it is possible to perform the refactoring, which is a transformation of the AST.

Program rendering. Once transformed, source code for the new program needs to be generated, conforming to the original program layout as much as possible.

Information gathering and transformation consist for the most part of 'boilerplate' code: generic operations are performed at the majority of AST nodes, with the real work being performed by *ad hoc* operations at particular kinds of node. These hybrid generic / specific traversals are supported by a number of systems: in HaRe we use Strafunski [LV01]; other systems include [LP03,LP04].

More details of the HaRe system, including the implementation of a particular refactoring and the details of program rendering, are given in Section 7 and the papers [LRT03,LRT04].

4 Structual refactorings

The first release of HaRe contained a number of refactorings which could be called *structural*. They principally concern the structure of a program, and in particular the objects defined, how they are named and what are their scopes. In summary, HaRe supports the following structural refactorings.

Delete a definition that is not used.

Duplicate a definition, under another name.

Rename a function, variable, type or any other program item.

Promote a definition from a local scope to a wider scope, or to the top level of the module.

Demote a definition which is only used within one definition to be local to that definition.

Introduce a definition to name an identified expression.

Add an argument to a function.

Remove an argument from a function, if it is not used.

Unfold a definition: in other words replace an occurrence of the left-hand side of a definition by the corresponding right-hand side.

Generalise a definition by making a selected sub-expression of its right-hand side into a value passed into the function via a new formal parameter.

A number of these refactorings are inverses of each other: promote / demote a definition; add / remove an argument. Others are not quite inverse; the principal example of this is the pair: unfold a definition / introduce a definition. Yet others have inverses yet to be implemented, including generalisation.

We look in more detail here at just one refactoring: generalisation, whose full catalogue entry is given in Figures 4 and 5. Note that, in common with many of these refactorings, generalisation has an effect throughout a module and indeed beyond, since both the definition of the function and all *calls* to the function must be modified.

Each refactoring is only valid under certain conditions. These conditions are covered in more detail in the paper [LRT03] and in the catalogue which accompanies the HaRe system [HaR].

Generalisation

Description: Generalise a definition by selecting a sub-expression of the right-hand side (here `"\n"`) of the definition and making this the value of a new formal parameter added to the definition of the function. The sub-expression becomes the actual parameter at all the call sites.

```
format :: [String] -> [String]
format []     = []
format [x]    = [x]
format (x:xs)
  = (x ++ "\n") : format xs

table = concat . format
```

```
format :: [a] -> [[a]] -> [[a]]
format sep []     = []
format sep [x]    = [x]
format sep (x:xs)
  = (x ++ sep) : format sep xs

table = concat . format "\n"
```

General comment: The choice of the position where the argument is added is not accidental: putting the argument at the beginning of the argument list means that it can be added correctly to any partial applications of the function. Note that in the Add Argument refactoring we name the new parameter at the same level as the definition, whereas here we substitute the expression at all call sites.

Left to right comment: In the example shown, a single expression is selected. It is possible to abstract over a number of occurrences of the (syntactically) identical expression by preceding this refactoring by

- a transformation to a single equation defined by a case expression;
- the introduction of a local definition of a name for the common expression.

and by following the refactoring by the appropriate inverse refactorings.

In a multi-module system, some of the free variables in the selected subexpression might not be accessible to the call sites in some client modules. Instead of explicitly exporting and/or importing these variables, the refactorer creates an auxiliary function (`fGen`, say) in the module containing the definition to represent the subexpression, and makes it accessible to the client modules.

Right to left comment: The inverse can be seen as a sequence of simpler refactorings.

- A definition of a special case is introduced: `fmt = format "\n"` and any uses of `format "\n"` (outside its definition) are folded to `fmt`.
- Using *generative* folding, the definition of `format` is specialised to a definition of `fmt`. (Folds in the style of Burstall and Darlington are called generative as they will generate a new definition.)
- If all uses of `format` take the parameter `"\n"` then no uses of `format` remain. Its definition can be removed, and `fmt` can be renamed to `format`.

(cont.)

Fig. 4. Catalogue entry for generalisation (part 1)

Left to right conditions: There are two conditions on the refactoring.

- Adding the new formal parameter should not capture any existing uses of variables.
- The abstracted sub-expression, `e` say, becomes the first argument of the new function at every use of it. For every new occurrence of `e` it is a requirement that the bindings of all free identifiers within `e` are resolved in the same way that they are in the original occurence.

Right to left conditions: The successful specialisation depends upon the definition of the function to have a particular form: the particular argument to be removed has to be a constant parameter: that is, it should appear unchanged in every recursive call.
The definition of the original function can only be removed if it is only used in the specialised form.

Analysis required: Static analysis of bindings; call graph; module analysis. If the type declaration is to be modified, then type inference will be needed.

Fig. 5. Catalogue entry for generalisation (part 2)

The most complex conditions are centered on the *binding structure* of the program: that is, the association between uses of identifiers (function and variable names, types, constructor names and so forth) and their definitions. Two examples serve to illustrate the point:

- If a definition is moved from a local scope to the top level, it may be that some names move out of their scope: this could leave them undefined, or bound to a different definition.
- In the case of generalisation, a new formal parameter is added to the definition in question: this may also disturb the binding structure, capturing references to an object of the same name defined at the top level.

Capture can occur in two ways: the new identifier may be captured, as when `f` is renamed to `g`:

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

f x = ...                            g x = ...
```

or it may capture other uses, as when a local definition `f` is renamed to `g`:

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

g x = ...                            g x = ...
```

In the next section we explore the impact of modules on the refactoring process for Haskell.

5 Modules and module-aware refactorings

The second release of HaRe extends the first in two ways. The structural refactorings are themselves made *module aware*, that is they are extended to have an effect throughout a multi-module project rather than in a single module alone. Various refactorings for the module system are then introduced.

5.1 Module-aware refactorings

A Haskell module may import definitions from other modules, and re-export them and its own definitions for use in other modules. This has the effect of widening the scope of a definition from a single module to a set of modules. It may be imported just under its name, or in 'qualified' form as `Module.name`. An exhaustive, formal, description of the Haskell module system, developed as a part of the Programatica project, is given in [DJH02].

Returning to our example of generalisation, it is necessary to consider the expression which becomes the new actual parameter at every call site of the function, in every module where the function is used. This expression will use identifiers defined in its home module, and these will need to be accessible. Two options present themselves. First, it would be possible to export all these definitions to all modules using the function, but this has the disadvantage of cluttering up the namespace with extraneous definitions, as well as introducing the possibility of name clashes. Instead, we introduce a new name for the actual parameter in the home module, and export that value together with the generalised function.

The scope of multi-module refactorings is not, of course, universal. In the HaRe project, we build on the Programatica infrastructure, and so we use the Programatica notion of *project* as delimiting the scope of a refactoring. In many cases it is possible to mitigate the effect of refactorings on modules outside the project. For example, if a generalised function is going to be used outside the project, then it is possible to build a 'wrapper' module which exports the original function rather than the generalised version.

5.2 Module refactorings

HaRe supports a number of refactorings related to the module system.

Clean the import list, so that the only functions imported are ones that are used in the module.
Make an explicit list of those bindings used from each imported module.
Add and **remove** items from the export list.
Move a definition from one module to another.

Consider the process of moving a top level definition of `f` from module `A` to `B`. First, various conditions need to be satisfied if the move is to happen.

- `f` should not already be defined at the top level of `B`.
- The free variables in `f` should be accessible within the module `B`.
- The move should not create a circularity in the module dependencies.[3]

If the conditions are satisfied then the refactoring can be achieved by moving the definition from `A` to `B` with some follow-up actions.

- Modify the import/export lists in the modules `A` and `B` *and* the client modules of `A` and `B` as necessary.
- Change uses of `A.f` to `B.f` or `f` in all affected modules.
- Resolve any ambiguity that might arise.

Other refactorings within the module system include: moving a group of definitions, moving type, class and instance definitions, and merging and splitting modules.

6 Data-oriented refactorings

This section looks in more detail at a number of larger-scale, data-oriented, refactorings. It is characteristic of all of these that they are bi-directional, with the context determining the appropriate direction. Some of these refactorings are described in the case study of [TR03]. The section concludes with an overview of other, type-based, refactorings.

6.1 Concrete to abstract types

One of the principal attractions of almost all modern functional programming languages is the presence of pattern matching.[4] Pattern matching combines selection between alternatives and extraction of fields, allowing definitions of data-processing functions to follow the template provided by the `data` definition closely. Take the example of a binary tree:

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

The definition has two cases: a `Leaf` and a (recursive) `Node`. Correspondingly, a function to `flatten` a tree into a list has two clauses: the first deals with a leaf, and the second processes a node, recursively:

[3] Whilst Haskell 98 allows recursive modules, the reason for this restriction is the imperfect support for recursive modules provided by current Haskell implementations.

[4] Scheme is the main exception, and indeed even within the Scheme community it is taken for granted that pattern matching macros are used by scheme programmers in all but the most introductory of contexts.

```
module Tree (Tree, leaf, node, isLeaf, isNode, val, left, right) where

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

isLeaf (Leaf _) = True                  val (Leaf x) = x
isLeaf _        = False                 val (Node x _ _) = x

isNode (Node _ _ _) = True              left  (Node _ l _) = l
isNode _            = False             right (Node _ _ r) = r

leaf = Leaf
node = Node
```

Fig. 6. `Tree` as an abstract data type

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

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

The disadvantage of this approach is the concrete nature of the definition of `Tree`: in other words, the *interface* to the `Tree` type is given by a pair of constructors:

```
Leaf :: a -> Tree a
Node :: a -> Tree a -> Tree a -> Tree a
```

`Leaf` and `Node` are not only functions, but also can be used in patterns for the `Tree` type. Every `Tree` is built by applying these constructors, and any function over `Tree` can use pattern matching over its arguements.
The alternative is to make `Tree` an abstract type. The interface to an abstract type is a collection of functions. Discrimination between the various cases and selection of components needs now to be provided explicitly by functions. The code for this case is shown in Figure 6. The selector functions can also be defined using field names.

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

Each function defined using pattern matching needs to be redefined. Case discrimination is replaced by guards, and selction by explicit selectors (given in this case by labelled fields):

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

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

A refactoring of this sort is often preliminary to a change of representation of the `Tree` type; after the refactoring this can be achieved by changing the definition of the interface functions; no client functions need to be modified.

HaRe supports this refactoring by means of a number of elementary refactorings:

Add field names. Names are added to the fields of the data type. Names are chosen by the system, but these can be changed using the renaming refactoring.

Add discrimiators. By default, discriminators are named '`isCon`' for the constructor `Con`. If functions of this name already exist, other names are chosen.

Add constructors. Functions `con` corresponding to the constructor `Con` are introduced.

Remove nested patterns. A particular problem is pressented by patterns containing constructors from other datatypes. Using the `Tree` example again, consider the fragment

```
f (Leaf [x]) = x+17
```

in which a list constructor occurs within a pattern from the `Tree` datatype. We will have to replace this pattern with a variable, and thus we lose the list pattern match too. So, we need to deal with this *nested* pattern first, thus:[5]

```
f (Leaf xs) = case xs of
                [x] -> x+17
```

We leave it to readers to convince themsleves that other forms of nesting do not require this treatment.

Remove patterns. Patterns in the `Tree` type can now be eliminated in terms of the discriminators and selectors. Picking up the previous example, we will have

```
f t
  | isLeaf t = case (val t) of
                [x] -> x+17
```

Create ADT interface. Move the type definition into a separate file with an interface containing the selectors, discriminators and constructor functions.

Views [Wad87,B$^+$] give a mechanism for pattern matching to cohabit with type abstraction. It would be possible to augment the refactoring to include the appropriate view, and to retain pattern matching definitions whilst introducing type abstraction, if the revised proposal [B$^+$] were to be incorporated into Haskell.

[5] It may also be necessary to amalgamate a number of clauses before performing this step, since it is not possible to 'fall through' a `case` statement.

```
data Tree a
  = Leaf { val::a, flatten:: [a] } |
    Node { val::a, left,right::(Tree a), flatten::[a] }

leaf x     = Leaf x [x]

node x l r = Node x l r (x : (flatten l ++ flatten r))
```

Fig. 7. Memoising `flatten` in the data representation

6.2 Inside or out?

The abstraction of `Tree` in Section 6.1 gives a minimal interface to the type: values can be constructed and manipulated, but no other functions are included in the 'capsule' or module which delimits the type representation.

Arguably, more functions, such as `flatten` in our running example. might be included in the capsule. What are the arguments for and against this?

Inside. A function included in the capsule has access to the representation, and so can be defined using pattern matching. This may be unavoidable or more efficient if the interface does not export sufficient functionality.

Outside. A function defined outside the capsule need not be re-defined when the implementation changes, whereas a function inside must be redefined.

This refactoring extends 'move definition between modules' since the definition itself may also be transformed on moving in or out of the capsule.

6.3 Change of representation: memoisation

One reason for a change of representation is to support a more efficient representation of a data type. Suppose that `Tree`s are repeatedly flattened. There is then a case for including a field in the representation to contain this *memoised* value.

Once data of this sort is included in a type, it is imperative for the consistency of the data representation that the type is abstract, so that values are only constucted and manipulated by functions which preserve the *invariant* property that the particular field indeed represents the memoised value.

This transformation can be supported in a refactoring. The transformed version of the running example is shown in Figure 7. The example shows that the value is memoised in the fields named `flatten`. The `leaf` constructor establises the invariant, and `node` will preserve it.

Incidentally, the memoisation is lazy: the memoised function is as strict or lazy as the original function, so that it is possible, for example. to extract any finite portion of the `flatten` field of `bigTree = node 1 bigTree bigTree`.

```
data Expr
  = Epsilon | .... |
    Then Expr Expr |
    Star Expr

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

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

Fig. 8. Two `data` types of regular expressions

6.4 Constructor or constructor function?

Figure 8 shows two variants of a type of regular expressions. The left-hand definition makes `plus` syntactic sugar: it will be expanded out before any function over `Expr` can be applied, and definitions for regular expressions need not treat the `plus` case separately, so

```
literals (Plus e)
  = literals (Then e (Star e))
  = literals e `union` literals e
  = ...
```

On the other hand, with the right-hand definition it is possible to treat `Plus` explicitly, as in

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

However, it is not just possible but necessary to define a `Plus` case for every function working over the right-hand variant of `Expr`, thus requiring more effort and offering more opportunity for error.[6]

In any particular situation, the context will be needed to determine which approach to use. Note, however, that the transition from left to right can seen as a refactoring: the definitions thus produced may then be transformed to yield a more efficient version as is possible for the `literals` function.

6.5 Algebraic or existential type?

The traditional functional programming approach would represent a type of shapes as an algebraic type, as shown in the left-hand side of Figure 9. Each function defined over `Shape` will perform a pattern match over shape. Extending the type to include a new kind of shape – `Triangle`, say – will require that all functions have a case added to deal with a triangular shape.

[6] A more persuasive example for this transformation 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.

```
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)
```

```
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 Float

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

Fig. 9. Algebraic or existential type?

The traditional OO approach will use subclassing or 'polymorphism' (in the OO sense) to implement conditional code.[7] This style is also possible in a functional context, using a combination of type classes and existential types. Figure 9 shows how to achieve this for a type of shapes. It is argued that an advantage of the right-hand representation is that it makes extension of the `Shape` type simpler. To add a triangle shape it is necessary to add a new instance declaration; this single point in the program will contain all the necessary declarations: in this case calculations of the area and perimeter of a triangle.

This approach is not without its drawbacks, however. In the setting of Haskell 98 a full treatment of 'binary methods' becomes problematic. For example it is impossible to define `==` on the existential version of `Shape` using the standard definition by case analysis over the two arguments. Instead, it is necessary to convert shapes to a single type (e.g. via `show`) to turn a case analysis over *types* into a corresponding case over values.

Each representation will be preferable in certain circumstances, just as row-major and column-major array representations are appropriate for different algorithms.[8] The transformation from left to right can be seen as the result of a sequence of simpler refactorings:

[7] This is one of Fowler's [Fow00] refactorings: *Replace Conditional with Polymorphism.*

[8] The reference to array representations is no accident: we can see the two type definitions as presenting clauses of function definitions in row- and column-major form.

```
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
```

```
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 = (-)
```

Fig. 10. Layered data types

- introducing the algebraic 'subtypes' corresponding to the constructors of the original type: in this case `Circle` and `Rect`;
- introducing a class definition for the functions: here the class `Sh`;
- introducing the instance declarations for each 'subtype',
- and finally introducing the existential type: in this example, `Shape`.

6.6 Layered data types

Figure 10 illustrates two alternative representations of a data type of arithmetic expressions. The left-hand approach is the more straightforward: the different sorts of arithmetic expression 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, and each of which is treated in a similar way. Refactoring for 'common code extraction' can make this similarity explicit.

On the other hand, in the right-hand definition, 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`.

Note that moving between representations requires the transformation of all definitions that either use or return an `Expr`.

```
data Expr
  = Lit Integer |             -- Literal integer value
    Vbl Var |                 -- Assignable variables
    Add Expr Expr |           -- Expression addition: e1+e2
    Assign Var Expr           -- Assignment: x:=e

type Var = String


type Store = [ (Var, Integer) ]

lookup :: Store -> Var -> Integer
lookup st x = head [ i | (y,i) <- st, y==x ]

update :: Store -> Var -> Integer -> Store
update st x n = (x,n):st
```

Fig. 11. Expressions and stores

6.7 Monadification

It is commonplace for Haskell programs to incorporate computational effects of various sorts, such as input/output, exceptions and state. Haskell is a pure language, and it is not possible simply to add side effects to the system; instead, expressions with related actions are embedded in a *monad*.

A monad in Haskell is given by a constructor class, which abstracts away from certain computational details of evaluating expressions with associated effects. In its interface lie two functions: `return` which creates an expression with null side effects, and `>>=` which is used to sequence and pass values between two side effecting computations.

A natural step for the programmer is to begin by defining a pure program: one which does no IO, for instance, and later to add IO actions to the program. This necessitates bringing monads to the program. There are two distinct flavours of monadification:

- a non-monadic program is 'sequentialized' to make it monadic; this is the work of Erwig and his collaborators [ER04];
- a program with explicit actions – such as a state 'threaded' through the evaluation – is made into a program which explicitly uses the monadic operations `return` and `>>=`, or indeed their 'sugared' version, the `do` notation.

An example of what is required can be see in Figures 11 and 12. Figure 11 shows a type of side-effecting expressions, and a `store` type. An example of the side effects are seen in

```
y := (x := x+1) + (x := x+1)
```

```
eval :: Expr -> Store -> (Integer, Store)

eval (Lit n) st
  = (n,st)


eval (Vbl x) st
  = (lookup st x,st)



eval (Add e1 e2) st
  = (v1+v2, st2)
    where
    (v1,st1) = eval e1 st
    (v2,st2) = eval e2 st1

eval (Assign x e) st
  = (v, update st' x v)
    where
    (v,st') = eval e st
```

```
evalST :: Expr -> State Store Integer

evalST (Lit n)
  = do
    return n

evalST (Vbl x)
  = do
    st <- get
    return (lookup st x)

evalST (Add e1 e2)
  = do
    v1 <- evalST e1
    v2 <- evalST e2
    return (v1+v2)

evalST (Assign x e)
  = do
    v <- evalST e
    st <- get
    put (update st x v)
    return v
```

Fig. 12. Evaluating expressions with side-effects

Evaluating this expression in a store where `x` has the value 3 results in `y` being assigned 9: the first sub expression has the value 4, the second 5.

Figure 12 gives two versions of an evaluator for these expressions. On the left-hand side is an evaluator which passes the `Store` around explicitly. The key case is the evaluation of `Add e1 e2` where we can see that `e2` is evaluated in the store `st1`, which may have been modified by the evaluation of `e1`.

On the right-hand side is the monadic version of the code. How easy is it to transform the left-hand side to the right? It is a combination of unfolding and folding function definitions, combined with the transformation between a `where` clause and a `let`. Unfolding and folding of functions defined in `instance` declarations necessitates a type analysis in order to associate uses of identifiers with their definitions. Existing work on describing monad intoduction includes Erwig and Ren's *monadification* [ER04] and Lämmel's *monad introduction* [Läm00].

6.8 Other type and data refactorings

A number of structural refactorings apply equally well to types: it is possible to rename, delete or duplicate a `type` definition, for instance. Others apply specifically to types:

Introduce a `type` definition. Type synonyms make a program easier to read, but have no semantic implication.

Introduce a `newtype`. Oh the other hand, a `newtype` is a new type, rather than a new name for an existing type. The restrictions that Haskell 98 places on which types may be declared as instances of classes make it necessary to introduce `newtypes` for composite types as instances.

Other data-related refactorings include:

Enumerated type. Replace a finite set of constants with an enumerated type; that is a `data` type with a finite number of 0-ary constructors.

`Maybe` types. Convert a `Maybe` type to a list or an `Either`; these can be seen as transformations of particular monads, as can the conversion from (the constant functor) `Bool` to a `Maybe` type.

Currying and uncurrying. It is possible to group, ungroup and reorder argument lists to functions and types.

Algebraic types. Convert between tuples and one-constructor algebraic types; between homogeneous tuples and lists.

Type generalisation. A type definition may refer to a particular type, as the right-hand definition of `Expr` in Figure 10 refers to `BinOp`; this reference can become an additional parameter to the definition, with compensating changes to be made to the remainder of the program.

Some of these refactorings are already implemented in HaRe; others are being developed. The next section offers users the possibility of implementing refactorings for themselves.

7 Designing your own refactorings: the HaRe API

The HaRe system has a layered architecture, illustrated in Figure 13. It is a Haskell program, so ultimately depends on a Haskell compiler for implementation. The Programatica toolset [Hal03] provides the front end functionality, and the traversals and analyses are written using Strafunski [LV01].

7.1 The HaRe API

Using these libraries we have built other libraries of utilities for syntax manipulation: functions to collect all free identifiers in an expression, substitution functions and so forth.

Two library layers are necessary because of our need to preserve program layout and comments. In common with the vast majority of compilers, Programatica's abstract syntax tree (AST) omits comments, and contains only a limited amount of source code location information.

To keep track of complete comment and layout data we work with the token stream output by the lexical analyser, as well as the AST. When a program is modified we update both the AST and the token stream, and we output the

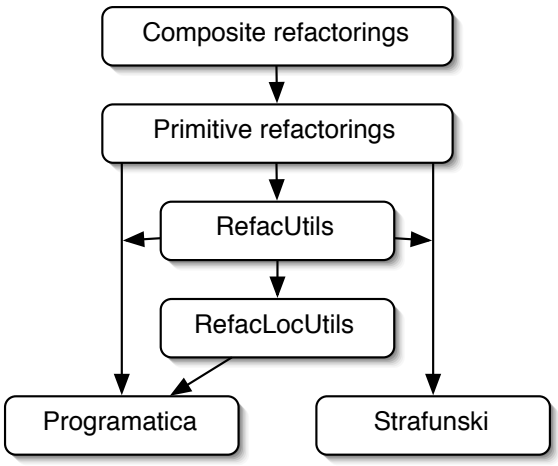

Fig. 13. The HaRe architecture

source code program by combining the information held by them both. This necessitates that the utilities we design must manipulate both AST and token stream; we provide two libraries to do this.

`RefacUtils`**:** this library hides the token stream manipulation, offering a set of high-level tree manipulation functions which will manipulate syntactic fragments; operations provided include insert, substitute, swap and so forth. These are built on top of our other library, which is described next.

`RefacLocUtils`**:** this library provides the functionality to manipulate the token stream and AST directly; it is the user's responsibility to maintain consistency between the two, whereas with `RefacUtils` this is guaranteed by the library.

In the general course of things we would expect the majority of users to work with `RefacUtils`.

7.2 A design example

An illustrative example is given by the refactoring which swaps the first two arguments of a function. The essence of this transformation is the function `doSwap`:

```
doSwap pn = applyTP (full_buTP (idTP `adhocTP` inMatch
                                `adhocTP` inExp
                                `adhocTP` inDecls
                               ))
```

```
inMatch ((HsMatch loc fun pats rhs ds)::HsMatchP)
      | pNTtoPN fun == pn
      = case pats of
         (p1:p2:ps) -> do
                           pats' <-swap p1 p2 pats
                           return (HsMatch loc fun pats' rhs ds)
         _          -> error "Insufficient arguments to swap."
inMatch m = return m

inExp exp@((Exp (HsApp (Exp (HsApp e e1)) e2))::HsExpP)
      | expToPN e == pn
      = swap e1 e2 exp
inExp e = return e
```

Fig. 14. Swapping arguments in a pattern match and a function application

a bottom-up tree transformation which is the identity except at pattern matches, function applications and type declarations. The `adhocTP` combinator of Strafunski is *ad hoc* in the sense that it applies its left hand argument except when the right hand one can be applied; the `TP` suffix denotes that this is a 'type preserving' traversal. The details of the expression and definition manipulation functions are in Figure 14. Note that `swap` will in general swap two syntactic fragments within an AST, and so will be usable in many contexts. The code in Figure 14 also illustrates the Programatica 'two level' syntax in action: the `Exp` constructors witness the recursive knot-typing.[9]

The code in Figure 14 makes the 'swap' transformation but raises an error at any point where the function is used with less than two arguments. In a full implementation this condition would be checked prior to applying the refactoring, with two possibilities when the condition fails.

- No action is taken unless all applications have at least two arguments.
- Compensating action is taken in cases with fewer arguments. In this case it is possible to replace these instances of a function `f`, say, with calls to `flip f`, where `flip f a b = f b a`. Note that in particular this handles 'explicit' applications of a function of the form `f $ a $ b`.

Full details of the API and function-by-function Haddock [Had] documentation are contained in the HaRe distribution. Details of implementing a number of fusion transformations are given in [Ngu04].

[9] The two-level syntax is exemplified by a definition of lists. First a type constructor function is defined, `data L a l = Nil | Cons a l` and then the recursive type is defined to be the fixed point of `L`, thus: `newtype List a = List (L a (List a))`. Since types cannot be recursive in Haskell, the fixed point introduces a wrapping constructor, `List` here. For example, under this approach the list `[2]` will be given by the term `List (Cons 2 (List Nil))`.

7.3 A domain-specific language for refactoring

Users who want to define their own refactorings can potentially interact with the system in two quite different ways. First, it is possible to build refactorings using the API already discussed; this required users to understand details of Strafunski, Haskell syntax in Programatica and the API Itself.

A simpler, but more limited, approach is to provide set of combining forms, or *domain-specific language*, for the existing refactorings, in analogy with the tactic (and tactical) languages of LCF-style proof assistants such as Isabelle[NPW02]. In more detail, users could be offered combinators for

sequencing a list of refactorings;
choice between a number of alternatives; and
repetition of a given refactoring, whilst it is applicable.

Examples of how the DSL might be used are already evident: lifting a definition to the top level (of a module) can be effected by repeatedly lifting it one level; the full ADT refactoring is given by sequencing a number of simpler operations. Building this DSL is a current research topic for the project.

8 Reflecting on refactoring

The work we have seen so far raises a number of questions and directions for future work.

8.1 The refactoring design space

In implementing refactorings it becomes apparent that a single refactoring can often have a number of variants, and it is not clear which of these should be implemented. We introduce the different sorts of variation through a series of examples.

All, one or some? In introducing a new definition by selecting an expression, should the definition just replace the single identified instance of the expression, all instances of that expression (in the module) or should the user be asked to select precisely those instances to be replaced?

Compensation or not? In lifting a definition to the top level, what should be done if there are bindings used in the definition which are not in scope at the top level? It is possible to compensate for this by adding extra parameters (λ-lifting), or the implementation may simply disallow this refactoring.

Modify or preserve? Generalisation as outlined in Section 4 modifies the generalised function itself (`format`), changing calls to the function throughout the project. An alternative is to define a generalised function, `format'` say, and to re-define `format` as an instance of this.

One advantage of this approach is that it localises the changes to a single module. Moreover the type of the function is unchanged, so that any uses of

the function outside the project will not be compromised. A disadvantage is the proliferation of names: the original function and its generalisation are both visible now.

How much to preserve? A refactoring should not change the behaviour of the program, but underneath this requirement lie a number of possibilities. It would be possible to require that the meaning of every definition was unchanged, but that would preclude a generalisation, which changes not only the function but its type as well.

More realistically, it would be possible to require that the meaning of the `main` program should be unchanged. This allows for a whole variety of refactorings which don't preserve meaning locally, but which do work 'under the hood'. To give two examples:

- The semantics of Haskell draws subtle distinctions between function bindings and lambda expressions, for instance, which are only apparent for partially-defined data values. Arguably these should not be allowed to obstruct transformations which can substantially affect program design.
- More problematic is a refactoring which replaces lists by sets, when the lists are essentially used to store a collection of elements.[10]

To verify the conditions of the last transformation in a tool would be a substantial challenge; this is the point at which a tool builder has to realise that the worth of the tool comes from implementing a set of clearly-defined, simple and useful refactorings, rather than attempting to be comprehensive.

Not (quite) a refactoring? Some operations on programs are not precisely refactorings, but can be supported by the same infrastructure, and would be of value to programmers. Examples include:

- Add a new constructor to a `data` type:[11] this should not only add the constructor but also add new clauses to definitions which use pattern matching over this type.
- Add a field to a constructor of a `data` type: this would require modification to every pattern match and use of this constructor.
- Create a new skeleton definition for a function over a `data` type: one clause would have to be introduced for each constructor.

8.2 What does the work say for Haskell?

Building a tool like HaRe makes us focus on some of the details of the design of Haskell, and how it might be improved or extended.

The correspondence principle. At first sight it appears that there are correspondences between definitions and expressions [Ten79], thus:

[10] This is illustrated in the case study [TR03].

[11] It is arguable that this is a refactoring, in fact. Adding a constructor only has an effect when that constructor is used, although this could arise indirectly through use of a derived instance of `Read`.

	Expressions	*Definitions*
Conditional	`if ... then ... else ...`	**guard**
Local definition	`let`	`where`
Abstraction	`\p -> ...`	`f p = ...`
Pattern matching	`case x of p ...`	`f p = ...`

In fact, it is not possible to translate freely between one construct and its correspondent. In general, constructs associated with definitions can be 'open ended' whereas expressions may not.

Take a particular case: a clause of a function may just define values for certain arguments because patterns or guards may not exhaust all the possibilities; values for other arguments may be defined by subsequent clauses. This is not the case with `if ... then ... else ...` and `case`: speaking operationally, once entered they will give a value for all possible arguments; it is not possible to fall through to a subsequent construct.

Arguably this reflects a weakness in the design of Haskell, and could be rectified by tightening up the form of definitions (compulsory `otherwise` and so forth), but this would not be acceptable to the majority of Haskell users.

Scoped instance declarations. In Haskell it is impossible to prevent an instance declaration from being exported by a module. The lack of scoped class instances is a substantial drawback in large projects. The specific difficulty we experienced was integrating libraries from Programatica and Strafunski which defined subtly different instances of the same class over the same type.

The module system. There are certain weaknesses in the module system: it is possible to hide certain identifiers on importing a module, but it is not possible to do the same in an export list, for instance.

Layout. In designing a tool which deals with source program layout, a major headache has been caused by tabs, and the different way in which they can be interpreted by different editors (and editor settings). We recommend that all users work with spaces in their source code.

Haskell 98 / GHC Haskell. Whilst we have built a system which supports full Haskell 98, it is apparent that the majority of larger-scale Haskell systems use the *de facto* standard, GHC Haskell. We hope to migrate HaRe to GHC in due course, particularly if we are able to use the GHC front end API currently under development.

8.3 An exercise for the reader

Readers who are interested in learning more about refactoring are encouraged to use the HaRe tool to support exploring refactoring in a particular project. Any non-trivial project would be suitable: the game of minesweeper [Tho] provides a nicely open-ended case study.

8.4 Future work

High on our priority list is to implement refactorings which will extract 'similar' pieces of program into a common abstraction: this is often requested by potential users. We also expect to migrate the system to deal with hierarchical libraries, libraries without source code and ultimately to use the GHC front end to support full GHC Haskell in HaRe.

8.5 Refactoring elsewhere

These notes have used Haskell as an expository vehicle, but the principles apply equally well to other functional languages, or at least to their pure subsets.

Programming is not the only place where refactoring can be useful. When working on a presentation, a proof, a set of tests and so forth similar redesigns take place. Formal support for proof re-construction could be added to proof assistants such as Isabelle.

In a related context, there is often interest in providing evidence for a program's properties or behaviour. This evidence can be in the form of a proof, test results, model checks and so forth. This raises the challenge of evolving this evidence as the program evolves, through both refactorings and changes of functionality.

References

B$^+$. Warren Burton et al. Views: An Extension to Haskell Pattern Matching. Proposed extension to Haskell; `http://www.haskell.org/development/views.html`.

Blu. The BlueJ system. `http://www.bluej.org/`.

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

DJH02. Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification for the Haskell 98 Module System. In *ACM Sigplan Haskell Workshop*, 2002.

Ecl. The Eclipse project. `http://www.eclipse.org/`.

ER04. Martin Erwig and Deling Ren. Monadification of functional programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

Flo79. Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8):455–460, 1979.

Fow00. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, 2000.

GN93. W.G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2, 1993.

Had. The Haddock documentation system for Haskell. `http:/www.haskell.org/haddock`.

Hal03. Thomas Hallgren. Haskell Tools from the Programatica Project (Demo Abstract). In *ACM Sigplan Haskell Workshop*, 2003.

HaR. The HaRe system. `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`.

Läm00. R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

LHJ95. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California*. ACM Press, 1995.

LP03. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the Workshop on Types in Language Design and Implementation*. ACM, 2003.

LP04. Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of International Conference on Functional Programming 2004*. ACM Press, 2004.

LRT03. Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In *ACM Sigplan Haskell Workshop*, 2003.

LRT04. Huiqing Li, Claus Reinke, and Simon Thompson. Progress on HaRe: the Haskell Refactorer. Poster presentation at the *International Conference on Functional Programming*, Snowbird, Utah. ACM, 2004.

LV01. Ralf Lämmel and Joost Visser. Generic Programming with Strafunski, 2001. `http://www.cs.vu.nl/Strafunski/`.

Ngu04. Chau Nguyen Viet. Transformation in HaRe. Technical report, Computing Laboratory, University of Kent, 2004. `http://www.cs.kent.ac.uk/pubs/2004/2021`.

NPW02. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof assistant for Higher-Order Logic*. Springer, 2002.

Opd92. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

Rus00. Daniel J. Russell. *FAD: Functional Analysis and Design Methodology*. PhD thesis, University of Kent, 2000.

Ten79. Robert D. Tennent. *Principles of Programming Languages*. Prentice Hall, 1979.

Tho. Simon Thompson. Minesweeper. `http://www.cs.kent.ac.uk/people/staff/sjt/craft2e/Games/`.

TR03. Simon Thompson and Claus Reinke. A Case Study in Refactoring Functional Programs. In *Brazilian Symposium on Programming Languages*, 2003.

UML. The Unified Modeling Language. `http://www.uml.org/`.

Wad87. Philip Wadler. Views: a way for pattern-matching to cohabit with data abstraction. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*. ACM Press, January 1987. (Revised March 1987).