Chapter 1

Formalisation of Haskell Refactorings

Huiqing Li¹, Simon Thompson¹

Abstract: Refactoring is a technique for improving the design of existing programs without changing their external behaviour. HaRe is the refactoring tool we have built to support refactoring Haskell 98 programs. Along with the development of HaRe, we have also investigated the formal specification and proof of validity of refactorings. This formalisation process helps to clarify the meaning of refactorings, improves our confidence in the behaviour-preservation of refactorings, and reduces the need for testing. This paper gives an overview of HaRe, and shows our approach to the formalisation of refactorings.

1.1 INTRODUCTION

Refactoring [3] is about improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (nor remove) any bugs. Separating general software updates into functionality changes and refactorings has well-known benefits. While it is possible to refactor a program by hand, tool support is considered invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the application of the refactoring itself, thus making refactoring less painful and less error-prone.

As part of our project ‘Refactoring Functional Programs’ [14], we have developed the Haskell Refactorer, HaRe [7], providing support for refactoring Haskell programs. HaRe covers the full Haskell 98 standard language, and is integrated with two development environments: Vim and (X)Emacs. Apart from preserving behaviour, HaRe preserves both the comments and layout of the refactored

¹¹Computing Laboratory, University of Kent, UK; Email: H.Li@kent.ac.uk, S.J.Thompson@kent.ac.uk
programs as much as possible. HaRe is itself implemented in Haskell. The first version of HaRe was released in October 2003, and since then more features have been added to make it more usable. By the third release of HaRe, it supports 24 refactorings, and also exposes an API [8] for defining refactorings or more general program transformations. The refactorings implemented in HaRe fall into three categories: structural refactorings, module refactoring, and data-oriented refactorings. Structural refactorings, such as generalising a definition, renaming a definition, unfold a definition and changing the scope of a definition, mainly concern the name and scope of the entities defined in a program and the structure of definitions; module refactorings, such as move a definition from one module to another, removing redundant imports, etc, concern the imports and exports of individual modules, and the relocation of definitions among modules; Data-oriented refactorings, such as from abstract to concrete data type, are associated with data type definitions. Apart from implementing the refactoring tool, we have also examined the formal specification and proof of validation of refactorings.

A number of tools [3] have been developed to automate the application of refactorings in the past decade, especially for object-oriented programming languages, however the study of formalisation and proof of refactorings has been lagging behind mostly due to the complexity of programming language semantics. Compared to imperative languages, pure functional languages have a stronger theoretical basis, and reasoning about programs written in pure functional languages is less complicated due to the referential transparency [4] property. This is also manifested by the collection of related work in the functional programming paradigm where functionality-preserving program transformations are used for reasoning about programs [13], for deriving efficient implementations from program specifications [1, 12], and for compiler optimisation [5]. This paper investigates the formal specification of refactorings as well as the proof of their functionality preservation within our context of refactoring. Two representative refactorings are examined in detail, and they are generalise a definition and move a definition from one module to another. The former, which is typical of the class
of *structural* refactorings, generalises a definition by making an identified expression of its right-hand side into a value passed into the function via a new formal parameter, therefore improves the usability of the definition, as illustrated by the trivial example shown in Figure 1.1, where the program before generalisation appears in the left-hand column and the program after generalisation appears in the right-hand column; the later, which is typical of the class of *module* refactorings, moves a top-level definition from its current module to a specified module. Together with the move of the definition is the modification to the imports/exports of the affected modules, which compensates for the changes caused by moving the definition, as shown in the example in Figure 1.2. By these two refactorings, we aim to illustrate how a number of *structural* refactorings and *module* refactorings can be formalised in a similar way. The formalisation of *data-oriented* refactorings and how type information can be used in the formalisation need further research, and are not covered in this paper.

For each refactoring, we give its formal definition consisting of the representation of the program before the refactoring, the side-conditions that should be met by the program in order for the refactoring to preserve behaviour and the representation of the program after the refactoring, and prove that the programs before and after the refactoring are equivalent in functionality under the given side-conditions.

While HaRe is targeted at Haskell 98, our first formalisation of refactorings is based on the simple \(\lambda\)-calculus augmented with letrec-expressions (denoted as \(\lambda_{\text{letrec}}\)). By starting from \(\lambda_{\text{letrec}}\), we could keep our specifications and proofs simple and manageable, but still reflect the basic characteristics of refactorings. In the case that a refactoring involves features not covered by \(\lambda_{\text{letrec}}\), such as data
constructors, the type system, etc, we could extend $\lambda_{\text{L}}\text{etrec}$ accordingly. Another reason for choosing $\lambda_{\text{L}}\text{etrec}$ is that although Haskell has been evolved to maturity in the last two decades, an officially defined, widely accepted semantics for this language does not exist yet.

The remainder of this paper is organised as follows. Section 1.2 gives an overview of the related work. Section 1.3 introduces $\lambda_{\text{L}}\text{etrec}$. Section 1.4 presents some definitions and lemmas needed for working with $\lambda_{\text{L}}\text{etrec}$ and for the specification of refactorings. Section 1.5 studies the formalisation of the generalise a definition refactoring. In Section 1.6, we extend $\lambda_{\text{L}}\text{etrec}$ to $\lambda_{\text{M}}$ to accommodate a simple module system. Some fundamental definitions with the simple module system are given in Section 1.7, and definitions are also used by the following formalisation of refactoring. Then the formalisation of the move a definition from one module to another refactoring is given in Section 1.8, and conclusions are drawn in Section 1.9.

1.2 RELATED WORK

Refactorings should preserve the behaviour of software. Ideally, the most fundamental approach is to formally prove that refactorings preserve the full program semantics. This requires a formal semantics for the target language to be defined. However, for most complex languages such as C++, it is very difficult to define a formal semantics. Even for a functional programming language like Haskell 98, there is still a lack of an officially defined semantics. In this case, people usually adopt the idea of invariants, pre-conditions or post-conditions, to ensure the preservation of semantics.

Opdyke [11] proposed a set of seven invariants to preserve behaviour for object-oriented programming language refactorings. These invariants are: unique superclass, distinct class names, distinct member names, inherited member variables not redefined, compatible signatures in member function redefinition, type-safe assignments and semantically equivalent reference and operations. Opdyke’s refactorings were accompanied by proofs which demonstrated that the enabling conditions he identified for each refactoring preserved the invariants. Opdyke did not prove that preserving these invariants preserves program behaviour. In [16], Tokuda et al. also made use of program invariants to preserve behaviour of refactorings.

In [15], F. Tip et al. explored the use of type constraints to verify the preconditions and to determine the allowable source code modifications for a number of generalisation related refactorings in an object-oriented program language context.

Using a different approach, Tom Mens et al. [10] explored the idea of using graph transformation to formalise the effect of refactorings and prove behaviour preservation in the object-oriented programming paradigm. This approach proposed to use graph to represent those aspects (access relation, update relation and call relation) of the source code that should be preserved by a refactoring, and graph rewriting rules as a formal specification for the refactoring transformations.
1.3 THE $\lambda$-CALCULUS WITH LETREC ($\lambda_{\text{LETREC}}$)

The syntax of $\lambda_{\text{LETREC}}$ terms is:

\[
\begin{align*}
V & ::= x \mid \lambda x. E \\
E & ::= V \mid E_1 E_2 \mid \text{letrec } D \text{ in } E \\
D & ::= \varepsilon \mid x_i = E_i \mid D, D
\end{align*}
\]

where $V$ represents the set of values, $E$ represents expressions, and $D$ is a sequence of bindings. A value is a variable or an abstraction. For letrec expressions, we require that the variables $x_i$ in the same binding sequence are pairwise distinct. Recursion is allowed in a letrec expression and the scope of $x_i$ in the expression, letrec $x_1 = E_1, \ldots, x_n = E_n$ in $E$, is $E$ and all the $E_i$\'s. No ordering among the bindings in a letrec expression is assumed. As a notation, we use $\equiv$ to represent syntactical equivalence, and $=$ to represent semantic equivalence.

As to the reduction strategy, one option for calculating lambda expressions with letrec is call-by-need [9], which is an implementation technique for the call-by-name [13] semantics that avoids re-evaluating expressions multiple times by memorising the result of the first evaluation. In the case that behaviour-preservation does not care about introducing/removing sharing of computation, strictly call-by-need might invalidate many refactorings which preserve the observable behaviour, but change the sharing of computation. Therefore, in this study, we use call-by-name for reasoning about program transformations, so that sharing could be lost or gained during the transformation. However, comments about the change of sharing during a refactoring will be given.

Instead of developing the call-by-name calculus for $\lambda_{\text{LETREC}}$ from scratch, we make use of the research results from the paper Lambda Calculi plus Letrec [17], in which Z. M. Ariola and S. Blom developed a call-by-name cyclic calculus ($\lambda_{\text{name}}$) and proved some results. $\lambda_{\text{name}}$ defines exactly the same set of terms as $\lambda_{\text{LETREC}}$ does, only with slightly different notation. Figure 1.3 lists the $\lambda_{\text{name}}$\'s axioms expressed using the $\lambda_{\text{LETREC}}$ notation of terms.

In the axioms shown in Figure 1.3, a $\tau$ attached to a term indicates that some bound variables in the term might be renamed to avoid name capture during the transformation; A context $C[]$ is a term with a hole in the place of one subterm. The two substitution axioms require that the $x$ in the hole occurs free in $C[x]$. $FV(E)$ means the set of free variables in term $E$. $D_1 \perp D_2$ means that the set of variables that occur as the left-hand side of an equation in $D$ does not intersect with the set of free variables of $D_2$. In the copying axiom, $\sigma$ is a function from recursion variable to recursion variables, and $E^{\sigma}$ is the term obtained by replacing all occurrence of recursion variables $x$ by $\sigma(x)$ (leaving the free variables of $E$ unchanged), followed by a reduction to normal form with the unification rule: $x = E, x = E \rightarrow x = E$ within the resulting letrec bindings.
Given an expression $E$ and a context $C$

**Definition 2**

Automatic renaming is defined as:

The result of substituting $N$ for the free occurrences of $x$ in $E$ with

**Definition 3**

Given two expressions $E$ and $E'$, $E'$ is a sub-expression of $E$ (notation $E' \subseteq E$), if $E' \subseteq \text{sub}(E)$, where sub($E$), the collection of sub-expressions of $E$, is defined inductively as follows:

- $\text{sub}(x) = \{ x \}$
- $\text{sub}(\lambda x. E) = \{ \lambda x. E \} \cup \text{sub}(E)$
- $\text{sub}(E_1 E_2) = \{ E_1, E_2 \} \cup \text{sub}(E_1) \cup \text{sub}(E_2)$
- $\text{sub}(\text{letrec } x_1 = E_1, ..., x_n = E_n \text{ in } E) = \{ \text{letrec } x_1 = E_1, ..., x_n = E_n \text{ in } E \} \cup \text{sub}(E) \cup \text{sub}(E_1) \cup ... \cup \text{sub}(E_n)$

**Definition 2**

Given an expression $E$ and a context $C[]$, we define sub($E, C$) as those sub-expressions of $C[E]$ which contain the hole filled with the expression $E$, that is: $e \in \text{sub}(E, C)$ iff $\exists C_1[], C_2[], ...$ such that $e \equiv C_2[E] \land C_1[] \equiv C_1[C_2[]]$.

**Definition 3**

The result of substituting $N$ for the free occurrences of $x$ in $E$ with automatic renaming is defined as:

$$x[x := N] = N$$

$$y[x := N] = y, \text{ where } y \neq x$$

**FIGURE 1.3.** The call-by-name cyclic calculus ($\lambda_{\text{name}}$) axioms

**1.4** THE FUNDAMENTALS OF $\lambda_{\text{LETREC}}$

**Definition 1**

Given two expressions $E$ and $E'$, $E'$ is a sub-expression of $E$ (notation $E' \subseteq E$), if $E' \subseteq \text{sub}(E)$, where sub($E$), the collection of sub-expressions of $E$, is defined inductively as follows:
Given an expression $i$ aliasing the definition $x$

Proof by induction on the structure of $E$.

Assume $E' = E$ where $(y \not\equiv x)$, and $y \equiv z$ if $x \not\in \text{FV}(E) \lor y \not\in \text{FV}(N)$, otherwise $z$ is a fresh variable.

Lettrec $x_1 = E_1, \ldots, x_n = E_n$ in $E_i[x := N]$

\[
(\text{letrec}\ x_1 = E_1[\bar{x} := \bar{z}][x := N], \ldots, z_n = E_n[\bar{x} := \bar{z}][x := N])
\]

where $z_i \equiv x_i$ if $x \not\in \text{FV}$ (letrec $x_1 = E_1, \ldots, x_n = E_n$ in $E$) $\lor x_i \not\in \text{FV}(N)$, otherwise $z_i$ is a fresh variable ($i=1..n$).

**Definition 4** Given $x \in \text{FV}(E)$ and a context $C[\ ]$, we say that $x$ is free over $C[E]$ only if $\forall e, e \in \text{sub}(E, C) \Rightarrow x \in \text{FV}(e)$. Otherwise we say $x$ becomes bound over $C[E]$.

**Lemma 1** Let $E'$, $E$ be expressions, and $E \equiv C[z]$, where $z$ is a free variable in $E$ and does not occur free in $C[\ ]$. If none of the free variables in $E'$ will become bound over $C[E']$, then $E[z := E'] \equiv C[E']$.

**Proof.** Proof by induction on the structure of $E$.

1.5 FORMALISATION OF GENERALISING A DEFINITION

1.5.1 Definition of generalise a definition

**Definition 5** Given an expression letrec $x_1 = E_1, \ldots, x_i = E_i, \ldots, x_n = E_n$ in $E_0$, assume $E'$ is a sub-expression of $E_i$, and $E_i \equiv C[E']$. Then the condition for generalising the definition $x_i = E_i$ on $E'$ is:

\[
x_i \not\in \text{FV}(E') \land \forall x, e : (x \in \text{FV}(E') \land e \in \text{sub}(E_i, C) \Rightarrow x \in \text{FV}(e)).
\]

After generalisation, the original expression becomes:

\[
\text{letrec}\ x_1 = E_1[x_i := x_i E'], \ldots, x_i = \lambda z.(C[z][x_i := x_i z]), \ldots, x_n = E_n[x_i := x_i E']
\]

in $E_0[x_i := x_i E'], \quad$ where $z$ is a fresh variable

What follows is some explanation about the above definition:

- The condition $x_i \not\in \text{FV}(E')$ means that there should be no recursive calls to $x_i$ within the identified sub-expression $E'$. Allowing recursive calls in the identified expression would need extra care to make sure the generalised function has the correct number of parameters at its call-sites.

- This specification replaces only the identified occurrence of $E'$ in the definition $x_i = E_i$. Another variant is to replace all the occurrences of $E'$ in $x_i = E_i$. This does not change the side-conditions for the refactoring, but it does change the transformation within $x_i = E_i$. According to this definition of generalising a definition, the refactoring could introduce duplicated computation. One way to avoid duplicating the computation of $x_i E'$ is to introduce a new binding for $x_i$. The choice of generalising only one occurrence is a trade-off between avoiding duplication and maintaining the correct number of parameters.
to represent the expression, instead of duplicating it at each call-site of \( x_i \). This also reflects the fact that under the same refactoring name, for instance \textit{generalising a definition}, different people may mean different things, and the choices is not unique.

1.5.2 Behaviour-preservation of \textit{generalising a definition}

In order to prove that this refactoring is behaviour-preserving, we decompose the transformation into a number of steps. If each step is behaviour-preserving, then we can conclude that the whole transformation is behaviour-preserving.

\textit{Proof.} Given the original expression:

\[
\text{letrec } x_1 = E_1, \ldots, x_i = E_i, \ldots, x_n = E_n \text{ in } E
\]

Generalising the definition \( x_i = E_i \) on the sub-expression \( E' \) can be decomposed into the following steps:

\textit{Step 1.} add definition \( x'_i = \lambda z. C[z] \), where \( x'_i \) and \( z \) are fresh variables, and \( C[E'] = E_i \), we get

\[
\text{letrec } x_1 = E_1, \ldots, x_i = E_i, x'_i = \lambda z. C[z], \ldots, x_n = E_n \text{ in } E
\]

The equivalence of semantics is guaranteed by the garbage collection rule and the commutability of bindings within \textit{letrec}.

\textit{Step 2.} By the side-conditions and axioms, we can prove

\[
x'_i E' \equiv (\lambda z. C[z]) E'_i
\]

\[
= \text{letrec } z = E'_i \text{ in } C[z] \quad \text{by } \beta \circ
\]

\[
= \text{letrec } z = E'_i \text{ in } C[E'_i] \quad \text{by substitution axiom and side-conditions}
\]

\[
= C[E'_i] \quad \text{by garbage collection axioms}
\]

\[
\equiv E_i
\]

Therefore replacing \( E_i \) with \( x'_i E' \) in the expression from step 1 does not change its semantics, and we have:

\[
\text{letrec } x_1 = E_1, \ldots, x_i = x'_i E'_i, x'_i = \lambda z. C[z], \ldots, x_n = E_n \text{ in } E
\]

\textit{Step 3.} Using the second \textit{substitution axiom}, it is trivial to prove that substituting \( x'_i E'_i \) for the free occurrences of \( x_i \) in the right-hand-side of \( x'_i \) does not change the semantics of \( x'_i \). We get

\[
\text{letrec } x_1 = E_1, \ldots, x_i = x'_i E'_i, x'_i = (\lambda z. C[z]) | x_i := x'_i E'_i, \ldots, x_n = E_n \text{ in } E
\]

As \( z \not\in \text{FV}(x'_i E') \), we have:

\[
\text{letrec } x_1 = E_1, \ldots, x_i = x'_i E'_i, x'_i = \lambda z. C[z] | x_i := x'_i E'_i, \ldots, x_n = E_n \text{ in } E
\]

8
Step 4. In the definition of $x_i'$, replace $E'$ with $z$. We get:

\[
\text{letrec } x_1 = E_1, \ldots, x_i = x_i'E', x_i' = \lambda z. C[z][x_i := x_i'z], \ldots, x_n = E_n \text{ in } E
\]

It is trivial to prove that the $x_i'$ defined in this step is not semantically equal to the $x_i'$ defined in step 3. However, we can prove the equivalence of $x_i'E'$ from step 3 to step 4 in the context of the bindings for $x_1, \ldots, x_n$ (note that $x_i'$ does not depend on the definition of $x_i$, so there is no mutual dependency between $x_i$ and $x_i'$). We omit the detailed proof for the lack of space, however, this should not affect the explanation of the overall approach.

Step 5. Substitute $x_i'E'$ for the free occurrences of $x_i$ outside the definition of $x_i$ and $x_i'$ does not change the semantics of the let-expression, as $x_i = x_i'(\alpha)$ from step 4.

\[
\text{letrec } x_1 = E_1[x_i := x_i'E'], \ldots, x_i = x_i'E', x_i' = \lambda z. C[z][x_i := x_i'z], \ldots, x_n = E_n[x_i := x_i'E'] \text{ in } E[x_i := x_i'E']
\]

Step 6. Remove the un-used definition of $x_i$ does not change the semantics, and we get

\[
\text{letrec } x_1 = E_1[x_i := x_i'E'], \ldots, x_i = \lambda z. C[z][x_i := x_i'z], \ldots, x_n = E_n[x_i := x_i'E'] \text{ in } E[x_i := x_i'E']
\]

Step 7. Rename $x_i'$ to $x_i$, we have

\[
\text{letrec } x_1 = E_1[x_i := x_i'E'][x_i' := x_i], \ldots, x_i = \lambda z. C[z][x_i := x_i'z][x_i' := x_i],
\ldots, x_n = E_n[x_i := x_i'E'][x_i' := x_i] \text{ in } E[x_i := x_i'E'][x_i' := x_i]
\]

Capture free renaming of bound variables, i.e. $\alpha$-renaming, does not change the semantics. Finally, by the substitution lemma, we have

\[
\text{letrec } x_1 = E_1[x_i := x_i'E'], \ldots, x_i = \lambda z. C[z][x_i := x_i'z], \ldots, x_n = E_n[x_i := x_i'E'] \text{ in } E[x_i := x_i'E']
\]

1.6 FORMALISATION OF A SIMPLE MODULE SYSTEM $\lambda_M$

A module-aware refactoring will affect not only the definitions in a module, but also the imports and exports of the module. More than that, it may potentially affect every module in the system. In order to formalise module-aware refactorings, we extend $\lambda_{\text{Letrec}}$ with a simple module system. The definition of the new language, which is called $\lambda_M$, is given next.
1.6.1 The Simple Module System $\lambda_M$

The syntax of $\lambda_M$ terms is defined as:

\[
\begin{align*}
Program & ::= \text{let } Mod \text{ in } (\text{Exp; Imp; letrec } D \text{ in } E) \\
Mod & ::= \epsilon | \text{Modid} = (\text{Exp; Imp; D}) | \text{Mod; Mod} \\
Exp & ::= \epsilon | (\text{Exp}^1, \ldots, \text{Exp}^n) \ (n \geq 0) \\
Exp' & = x \mid \text{Modid}.x \mid \text{module Modid} \\
Imp & ::= (\text{Imp}^1_1, \ldots, \text{Imp}^n) \ (n \geq 0) \\
Imp' & = \text{import Qual Modid Alias ImpSpec} \\
Modid & ::= M_i \ (i \geq 0) \\
Qual & ::= \epsilon | \text{qualified} \\
ImpSpec & ::= \epsilon \mid (x_1, \ldots, x_n) \mid \text{hiding } (x_1, \ldots, x_n) \ (n \geq 0) \\
Alias & ::= \epsilon \mid \text{as Modid} \\
V & ::= x \mid \text{Modid}.x \mid \lambda x.E \\
E & ::= V \mid E_1 E_2 \mid \text{letrec } D \text{ in } E \\
D & ::= \epsilon \mid x = E \mid D, D 
\end{align*}
\]

In the above definition, $Program$ represents a program and $Mod$ is a sequence of modules. Each module has a unique name in the program. A module consists of three parts: $Exp$, which exports some of the locally available identifiers for use by other modules; $Imp$, which imports identifiers defined in other modules, and $D$, which defines a number of value identifiers. The $(\text{Exp; Imp; letrec } D \text{ in } E)$ part in the definition of $Program$ represents the $\text{Main}$ module of the program, and the expression $E$ represents the main expression. $\epsilon$ means omitted export list and entity list respectively in the definitions of $Exp$ and $ImpSpec$, and empty in other definitions. Qualified names are allowed, and we assume that the usage of qualified names follows the rules specified in the Haskell 98 Report [6].

The module system has been defined to model the Haskell 98 module system. Because only value variables can be defined in $\lambda_M$, $\lambda_M$'s module system is actually a subset of the Haskell 98 module system. Therefore, we assume that the semantics of this simple module system follows the semantics of the Haskell 98 module system.

A formal specification of the Haskell 98 module system has been described in [2], where the semantics of a Haskell program with regard to the module system is a mapping from the collection of modules to their corresponding in-scope and export relations. The in-scope relation of a module represents the set of names (with the represented entities) that are visible to this module, and this forms the top-level environment of the module. The export relation of a module represents the set of names (also with the represented entities) that are made available for other modules to use by this module; in other words, it defines the interface of the module.

In the following specification of module-aware refactorings, we assume that,
by using the module system analysis algorithm from the formal specification given in [2], we are able to get the in-scope and export relation of each module, and for each identifier in the in-scope/export relation, we can infer the name of the module in which the identifier is defined. In fact, the same module analysis system is used by HaRe in its implementation.

When only module-level information is relevant, i.e. the exact definition of entities is not of concern, we can view a multi-module program in this way: a program P consists of a set of modules and each module consists of four parts: the module name, \( M \), the set of identifiers defined by this module, \( D \), the set of identifiers imported by this module, \( I \), and the set of identifiers exported by this module, \( E \). Each top-level identifier can be uniquely identified by the combination of the identifier’s name and its defining module as \((modid, id)\), where \( modid \) is the name of the identifier’s defining module and \( id \) is the name of the identifier. Two identifiers are the same if they have the same name and defining module. Accordingly, we can use \( P = \{(M_i, D_i, I_i, E_i)\}_{i=1..n} \) to denote the program.

1.7 FUNDAMENTALS OF \( \lambda_M \)

**Definition 6** A client module of module \( M \) is a module which imports \( M \) either directly or indirectly; A server module of module \( M \) is a module which is imported by module \( M \) either directly or indirectly.

**Definition 7** Given a module \( M=(\text{Exp}, \text{Imp}, D) \), we say module \( M \) is exported by itself if \( \text{Exp} \in \varepsilon \) or module \( \text{M} \) occurs in \( \text{Exp} \) as an element of the export list.

**Definition 8** The defining module of an identifier is the name of the module in which the identifier is defined.

**Definition 9** Suppose \( v \) is an identifier that is in scope in module \( M \), we use \( \text{defineMod}(v, M) \) to represent the name of the module in which the identifier is defined.

**Definition 10** We say that the identifier x defined in module \( N \) is used by module \( M=(\text{Exp}, \text{Imp}, D) \) \( (M \neq N) \) if \( \text{DefineMod}(x, M) = N \) and either \( x \in \text{FV}(D) \) or \( x \) is exported by module \( M \), otherwise we say that the x defined in module \( N \) is not used by module \( M \).

**Definition 11** Binding structure refers to the association of uses of identifiers with their definitions in a program. Binding structure involves both top-level variables and local variables. When analysing module-level phenomena, it is only the top-level bindings that are relevant. When only top-level identifiers are concerned, we define the binding structure, \( B \), of a program \( P = \{(M_i, D_i, I_i, E_i)\}_{i=1..n} \) as: \( B \subset \bigcup \{D_i \times (D_i \cup I_i)\}_{i=1..n} \times \{\{(m_1, id_1), (m_2, id_2)\} \in B| id_2 \text{ occurs free in the definition of id}_1; id_1 \text{'s defining module is } m_1, \text{ and id}_2 \text{'s defining module is } m_2 \} \).

**Definition 12** Given a set of identifiers \( Y \) and an export list \( \text{Exp} \), \( \text{rmFromExp}(\text{Exp}, Y) \) removes the occurrences of the identifiers of \( Y \) from \( \text{Exp} \), and is defined as:
\[ \text{rmFromExp} (\varepsilon, Y) = \varepsilon \]
\[ \text{rmFromExp} ((,), Y) = () \]
\[ \text{rmFromExp} (((e, \text{Exp}'_2, \ldots, \text{Exp}'_n), Y) \ (e \notin Y) \]
\[ = (e, (\text{rmFromExp}(\text{Exp}'_2, \ldots, \text{Exp}'_n), Y)) \]
\[ \text{rmFromExp} (((e, \text{Exp}'_2, \ldots, \text{Exp}'_n), Y) \ (e \in Y) \]
\[ = \text{rmFromExp} ((\text{Exp}'_2, \ldots, \text{Exp}'_n), Y) \]

Similar to the above definition, the following four definitions also involve syntactical manipulation of the import/export list. Due to the space limit, we give their descriptions, but omit the concrete definitions.

**Definition 13** Given an identifier \( y \) which is defined in module \( M \), and the export list, \( \text{Exp} \), of module \( M \), \( \text{addToExp}(\text{Exp}, y, M) \) ensures that module \( M \) exports \( y \).

**Definition 14** Given an identifier \( y \) which is exported by module \( M \) and \( \text{Imp} \) which is a sequence of imports, \( \text{rmFromImp}(\text{Imp}, y, M) \) removes the literal occurrences of \( y \) in the import declarations that import \( M \). The function can be used to remove the uses of \( y \) in import declarations that import module \( M \) when \( y \) is no longer exported by \( M \).

**Definition 15** Given an identifier \( y \) which is exported by module \( M \) (\( M \) is not necessarily the module where \( y \) is defined) and \( \text{Imp} \) which is a sequence of imports, then \( \text{hideInImp}(\text{Imp}, y, M) \) ensures that \( \text{Imp} \) does not bring this identifier into scope by importing it from module \( M \).

**Definition 16** Suppose the same binding, say \( y \), is exported by both module \( M_1 \) and \( M_2 \), and \( \text{Imp} \) is a sequence of import declarations, then \( \text{chgImpPath}(\text{Imp}, y, M_1, M_2) \) switches the importing of \( y \) from \( M_1 \) to \( M_2 \).

### 1.8 FORMALISATION OF MOVE A DEFINITION FROM ONE MODULE TO ANOTHER IN \( \lambda_M \)

Like other refactorings, the realisation of *Move a definition from one module to another* is non-unique. Suppose we would like to move the definition of \( \text{foo} \) from module \( M \) to module \( N \), the following design decisions were made during the implementation of this refactoring in HaRe.

- If a variable which is free in the definition of \( \text{foo} \) is not in scope in module \( N \), then the refactorer will ask the user to refactor the program to make the variable visible to module \( N \) first.
- If the identifier \( \text{foo} \) is already in scope in module \( N \) (either defined by module \( N \) or imported from other modules), but it refers to \( \text{foo} \) other than the one defined in module \( M \), the user will be prompted to do renaming first.
- We avoid introducing mutually recursive modules during a refactoring due to the fact that transparent compilation of mutually recursive modules are not yet supported by the current working Haskell compilers/interpreters.
• Module \( N \) will export \( \text{foo} \) after the refactoring only if \( \text{foo} \) is either exported by module \( M \) or used by the other definitions in module \( M \) before the refactoring. The imports of \( \text{foo} \) will be via \( M \) if module \( M \) still exports \( \text{foo} \) after the refactoring; otherwise via \( N \).

1.8.1 Definition of move a definition from one module to another

Next is the definition of move a definition from one module to another. A commentary on the definition follows.

**Definition 17** Given a valid program \( P \):

\[
P = \text{let } M_1 = (\text{Exp}_1; \text{Imp}_1; x_1 = E_1, \ldots, x_i = E_i, \ldots, x_n = E_n); \]
\[
M_2 = (\text{Exp}_2; \text{Imp}_2; D_2); \ldots; M_m = (\text{Exp}_m; \text{Imp}_m; D_m)
\]

in \( (\text{Exp}_0; \text{Imp}_0; \text{letrec } D_0 \text{ in } E) \)

The conditions for moving the definition \( x_i = E_i \) from module \( M_1 \) to another module, \( M_2 \), are:

1. If \( x_i \) is in scope at the top level of \( M_2 \), then \( \text{DefineMod}(x_i, M_2) = M_1 \).
2. \( \forall \ v \in \text{FV}(x_i = E_i) \), if \( \text{DefineMod}(v, M_1) = \text{N} \), then \( v \) is in scope in \( M_2 \) and \( \text{DefineMod}(v, M_2) = \text{N} \).
3. If \( M_1 \) is a server module of \( M_2 \), then \( \{x_i, M_1, x_i\} \cap \text{FV}(E_j, j \neq i) = \emptyset \).
4. If module \( M_{j(j \neq 1)} \) is a server module of \( M_2 \), and \( x_i \in \text{FV}(D_j) \), then \( \text{DefineMod}(x_i, M_j) \neq M_1 \) (\( x_i \) could be qualified or not).

After moving the definition to module \( M_2 \), the original program becomes:

\[
P' = \text{let } M_1 = (\text{Exp}_1'; \text{Imp}_1'; x_1 = E_1, \ldots, x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, \ldots, x_n = E_n); \]
\[
M_2 = (\text{Exp}_2'; \text{Imp}_2'; x_i = E_i[M_1, x_i := M_2, x_i], D_2); \ldots; \]
\[
M_m = (\text{Exp}_m; \text{Imp}_m'; D_m)
\]

in \( (\text{Exp}_0; \text{Imp}_0; \text{letrec } D_0 \text{ in } E) \)

The specification of the imports/exports is given according to whether \( x_i \) is exported by \( M_1 \), and different situations are considered in each case. To save space, only those imports/exports which are affected by the refactoring are given.

Case 1. \( x_i \) is not exported by \( M_1 \).
Case 1.1. \( x_i \) is not used by other definitions in \( M_1 \): \( \{x_i, M_1, x_i\} \cap \text{FV}(E_{j(j \neq i)}) = \emptyset \).
\( \text{Imp}_j' = \text{hideInImp} (\text{Imp}_j, x_i, M_2) \) if \( M_2 \) is exported by itself;
\( \text{Imp}_j' \) otherwise. (\( 3 \leq j \leq m \))

Case 1.2. \( x_i \) is used by other definitions in \( M_1 \).
\( \text{Imp}_1' = \text{hideInImp} (\text{Imp}_1, x_i, M_2); \) import \( M_2 \) as \( M_1(x_i) \)
\[
\begin{align*}
\text{Exp}'_2 &= \text{addToExp} (\text{Exp}_2, x_i, M_2) \\
\text{Imp}'_j &= \text{hideInImp} (\text{Imp}_j, x_i, M_2) \quad (3 \leq j \leq m)
\end{align*}
\]

Case 2. \(x_i\) is exported by \(M_1\).

Case 2.1. \(M_2\) is not a client module of \(M_1\).
\[
\begin{align*}
\text{Imp}'_1 &= \text{Imp}_1; \text{import} M_2 \text{ as } M_1 (x_i) \\
\text{Exp}'_2 &= \text{addToExp} (\text{Exp}_2, x_i, M_2) \\
\text{Imp}'_j &= \text{hideInImp} (\text{Imp}_j, x_i, M_2) \quad (3 \leq j \leq m)
\end{align*}
\]

Case 2.2. \(M_2\) is a client module of \(M_1\).
\[
\begin{align*}
\text{Exp}_1' &= \text{rmFromExport} (\text{Exp}_1, x_i, M_1) \\
\text{Exp}_2' &= \text{addToExp} (\text{Exp}_2, x_i, M_2) \\
\text{Imp}_2' &= \text{rmFromImp} (\text{Imp}_2, x_i, M_1) \\
\text{Imp}_j' &= \text{if } M_j \text{ is a server module of } M_2 \text{ then } \text{rmFromImp} (\text{Imp}_j, x_i, M_1) \\
&\quad \text{else } \text{rmFromImp} (\text{chgImportPath} (\text{Imp}_j', x_i, M_1, M_2), x_i, M_1) \quad (3 \leq j \leq m) \\
\text{Imp}_j'' &= \text{if } x_i \text{ is exported by } M_2 \text{ before refactoring, then } \text{Imp}_j; \\
&\quad \text{hideInImp} (\text{Imp}_j, x_i, M_2) \text{ otherwise.} \quad (3 \leq j \leq m)
\end{align*}
\]

What follows is some explanation about the above definition:

- As to the side-conditions, condition 1) means that if \(x_i\) is in scope in the target module, \(M_2\), then this \(x_i\) should be the same as the \(x_i\) whose definition is to be moved, in order to avoid conflict/ambiguous occurrence [6] in \(M_2\); condition 2) requires that all the free variables used in the definition of \(x_i\) are in scope in \(M_2\). Conditions 3) and 4) guarantee that mutual recursive modules won’t be introduced during the refactoring process.

- The transformation rules are complicated mainly due to the Haskell 98 module system’s lack of control in the export list. For example, when a new identifier is brought into scope in a module, the identifier could also be exported automatically by this module, and then further exported by other modules if this module is imported and exported by those modules. However, this is dangerous in some cases as the new entity could cause name conflict/ambiguity in modules which import it either directly or indirectly. Two strategies are used in the transformation in order to overcome this problem: the first strategy is to use hiding to exclude an identifier from being imported by another module when we are unable to exclude it from being exported, as in case 1.1; the second strategy is to use alias in the import declaration to avoid the changes to the export list as in case 1.2.

1.8.2 Behaviour-preservation of move a definition from one module to another

We prove the correctness of this refactoring from four aspects: the refactoring does not change the structure of individual definitions; the refactoring creates a binding structure which is isomorphic to the one before the refactoring; the
refactoring does not introduce mutually recursive modules; and the refactoring
does not violate any syntactical rules. More details follow.

- The refactoring does not change the structure of individual definitions. This
  is obvious from the transformation rules. Inside the definition of \( x_i = E_i \),
  the uses of \( M_1.x_i \) have been changed to \( M_2.x_i \), this is necessary as \( x_i \)
  is now defined in module \( M_2 \). We keep the qualified names qualified in order to
  avoid name capture inside the \( E_i \). The uses of \( x_i \) in module \( M_2 \) will not cause
  ambiguous reference due to condition a).

- The refactoring creates a binding structure which is isomorphic to the one
  before the refactoring. Suppose the binding structures before and after the
  refactoring are \( B \) and \( B' \) respectively, then \( B \) and \( B' \) satisfy:

\[
B' = \{(fx, fy) | (x, y) \in B\},
\]

where \( f(M, x) = (M_2, x_i) \) if \( (M, x) \equiv (M_1, x_i) \); \( (M, x) \) otherwise.

The only change from \( B \) to \( B' \) is that the defining module of \( x_i \) has been
changed from the \( M_1 \) to \( M_2 \). This is guaranteed by conditions a) and b).

- The refactoring does not introduce recursive modules. On one hand, moving
  the definition does not add any import declarations to \( M_2 \), therefore, there is
  no chance for \( M_2 \) to import any of its client modules. On the other hand, an
  import declaration importing \( M_2 \) is added to other modules only when it is
  necessary and \( M_2 \) is not a client module of them because of conditions c), d)
  and the condition checking in case 2.2.

- The refactoring does not violate any syntactical rules. The only remaining
  potential violations exist in the import/export lists of the modules involved. In
  case 1.1, case 1.2 and case 2.1, except module \( M_2 \), none of the modules’ in
  scope/export relations have been changed; in case 2.2, \( M_1 \) no longer exports \( x_i \),
  and those modules which use \( x_i \) now get it from module \( M_2 \). \( rmFromExport, \)
  \( addToExp \), \( rmFromImp \), and \( hideInImport \) are used to make manipulate the
  program syntactically to ensure the program’s syntactic correctness.

1.9 CONCLUSIONS AND FUTURE WORK

This paper explores the formal specification and proof of behaviour-preservation
of refactorings in the context of refactoring Haskell programs. To this purpose,
we first defined the simple lambda-calculus called \( \lambda_{Letrec} \), then augmented it with
a simple module system. Two representative refactorings are examined in this
paper, and they are _generalise a definition_ and _move a definition from one module
to another_. We feel that this work can serve as a starting point for further study of
formalising the essence of Haskell refactoring. For future work, more structural
or module-related refactorings, such as _renaming_, _specialise a definition_, _lifting
a definition_, _add an item to the export list_, etc [14], can be formalised in this
framework without difficulty. This framework can be extended to accommodate
more features from the Haskell 98 language, such as constants, case-expressions,
data types, etc, so that more complex refactorings can be formalised.
REFERENCES


