

Formalisation of Haskell Refactorings

Huiqing Li* and Simon Thompson**

Computing Laboratory, University Of Kent, UK

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 INTRODUCTION

Refactoring [8] 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. If applied properly, refactoring can make a program easier to understand or modify. 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’ [16], we have developed the Haskell Refactorer, HaRe [12], providing support for refactoring Haskell programs. HaRe covers the full Haskell 98 standard language, and is integrated with two development environments: Vim [2] and (X)Emacs [1, 3]. Apart from preserving behaviour, HaRe preserves both the comments and layout of the refactored 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 [13] for defining refactorings or more general program transformations.

Apart from implementing the refactoring tool, we have also examined the formal specification and proof of validation of refactorings. 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

*H.Li@kent.ac.uk

**S.J.Thompson@kent.ac.uk

to the referential transparency [9] 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 [15], for deriving efficient implementations from program specifications [5, 6, 14], and for compiler optimisation [10].

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*. 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 [11], our first formalisation of refactorings is based on the simple λ -calculus augmented with let-expressions (denoted as λ_{Letrec}). By starting from λ_{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 λ_{Letrec} , such as data constructors, the type system, etc, we could extend λ_{Letrec} accordingly. Another reason for choosing λ_{Letrec} 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 2 introduces λ_{Letrec} . Section 3 presents some definitions and lemmas needed for working with λ_{Letrec} . Section 4 studies the formalisation of the *generalise a definition* refactoring. In Section 5, we extend λ_{Letrec} to λ_M to accommodate a simple module system. Some fundamental definitions with the simple module system are given in Section 6. Then the formalisation of the *move a definition from one module to another* refactoring is given in Section 7, and conclusions are drawn in Section 8.

2 THE λ -CALCULUS WITH LETREC (λ_{LETREC})

The syntax of λ_{Letrec} terms is:

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

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** $x_1 = E_1, \dots, x_n = E_n$ 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, we could choose either the call-by-name [15] reduction or the call-by-need reduction. Call-by-need [4] is an implementation technique for a call-by-name semantics that avoids re-evaluating expressions multiple times by memorising the result of the first evaluation. Hence, call-by-need provides the ability to reason about sharing.

The advantage of using call-by-need is that it allows to detect the change of sharing during refactoring. However, call-by-name allows for both removing and introducing sharing during program transformations, and is simpler and more straightforward to use. In our proof, we use a calculus that permits transformations that are not strictly call-by-need, but a mixture of call-by-need and call-by-name, so that sharing could be removed or introduced. However comments about the change of sharing during a refactoring will be given.

The following reduction rules of λ_{Letrec} are based on the call-by-need calculus proposed by Z. M. Ariola, et al. in [4]. The most obvious difference between λ_{Letrec} and the call-by-need calculus proposed in [4] is in rule (1) where substitution, rather than a letrec expression (i.e. $\text{letrec } x = E_1 \text{ in } E$), is used to reduce a lambda application expression. The reduction rules for λ_{Letrec} are:

$$(\lambda x. E) E_1 = E[x := E_1] \quad (1)$$

$$(\text{letrec } D \text{ in } V) E = \text{letrec } D' \text{ in } V' E \quad (2)$$

$$\text{letrec } D_1 \text{ in } (\text{letrec } D \text{ in } E) = \text{letrec } D_1, D' \text{ in } E' \quad (3)$$

$$\text{letrec } x = V, D \text{ in } EC[x] = \text{letrec } x = V, D \text{ in } EC[V] \quad (4)$$

$$\begin{aligned} \text{letrec } x_1 = V, x_2 = EC_2[x_1], \dots, x_j = EC_j[x_{j-1}], D \text{ in } EC[x_j] \\ = \text{letrec } x_1 = V, x_2 = EC_2[V], \dots, x_j = EC_j[x_{j-1}], D \text{ in } EC[x_j] \end{aligned} \quad (5)$$

$$\text{letrec } x = EC_1[x], D \text{ in } EC[x] = \text{letrec } x = EC'_1[EC_1[x]], D \text{ in } EC[x] \quad (6)$$

$$\text{letrec } x = E_1 \text{ in } E = E, \text{ if } x \notin FV(E) \quad (7)$$

$$\begin{aligned} \text{letrec } x = E_1, x_2 = E_2, \dots, x_n = E_n \text{ in } E \\ = \text{letrec } x_2 = E_2, \dots, x_n = E_n \text{ in } E, \text{ if } x \notin FV(E) \wedge x \notin FV(E_{i(i=2..n)}) \end{aligned} \quad (8)$$

In the above rules, $E[x := E_1]$ stands for a capture avoiding substitution of E_1 for each free occurrence of x in E as defined by definition 6 in Section 3; a $'$ attached to a term indicates that some bound variables in the term might be renamed to avoid name capture during the transformation; $FV(E)$ means the set of free variables in expression E as defined by definition 2 in Section 3, and EC is used to represent evaluation contexts. An evaluation context is a context with a single hole which indicates the place at which the next expression is to be reduced. The evaluation contexts for λ_{Letrec} are defined as:

$$\begin{aligned} EC ::= [] \mid EC E \mid \text{letrec } D \text{ in } EC \\ \mid \text{letrec } x_1 = EC_1, \dots, x_j = EC_j[x_{j-1}], D \text{ in } EC[x_j] \end{aligned}$$

For the λ_{Letrec} calculus, we assume the presence of the following axiom and inference rules that makes provable equality a congruence relation.

$$E = E \quad (9)$$

$$E_1 = E_2 \Rightarrow E_2 = E_1 \quad (10)$$

$$E_1 = E_2 \Rightarrow E_2 = E_3 \Rightarrow E_1 = E_3 \quad (11)$$

$$E_1 = E_2 \Rightarrow C[E_1] = C[E_2] \quad (12)$$

$$\text{letrec } \varepsilon \text{ in } E = E \quad (13)$$

where $C[\]$ represents an expression with one hole in it as defined in Section 3.

3 THE FUNDAMENTALS OF λ_{LETREC}

This section introduces some definitions and lemmas working with λ_{Letrec} .

Definition 1 Given two expressions E and E' , E' is a sub-expression of E (notation $E' \subseteq E$), if $E' \in \text{sub}(E)$, where $\text{sub}(E)$, the collection of sub-expressions of E , is defined inductively as follows:

$$\begin{aligned} \text{sub}(x) &= \{x\} \\ \text{sub}(\lambda x . E) &= \{\lambda x . E\} \cup \text{sub}(E) \\ \text{sub}(E_1 E_2) &= \{E_1 E_2\} \cup \text{sub}(E_1) \cup \text{sub}(E_2) \\ \text{sub}(\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E) &= \\ &= \{\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E\} \cup \text{sub}(E) \cup \text{sub}(E_1) \cup \dots \cup \text{sub}(E_n) \end{aligned}$$

Definition 2 $FV(E)$ is the set of free variables in expression E , and is defined as:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x . E) &= FV(E) - \{x\} \\ FV(E_1 E_2) &= FV(E_1) \cup FV(E_2) \\ FV(\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E) &= \\ &= (FV(E_1) \cup \dots \cup FV(E_n) \cup FV(E)) - \{x_1, \dots, x_n\} \end{aligned}$$

Definition 3 $TBV(E)$ is the set of variables which are bound at the top level of E and can be defined as:

$$\begin{aligned} TBV(x) &= \{ \} \\ TBV(\lambda x . E) &= \{x\} \\ TBV(E_1 E_2) &= \{ \} \\ TBV(\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E) &= \{x_1, \dots, x_n\} \end{aligned}$$

Definition 4 A Context $C[\]$ is an expression with one hole in it; A multi-place context $M[\]$ is an expression with none, one or more holes in it.

Definition 5 Given an expression E and a context $C[\]$, we define $\text{sub}(E, C)$ 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] \wedge C[\] \equiv C_1[C_2[\]]$.

Definition 6 The result of substituting N for the free occurrences of x in E with automatic renaming is defined as:

$$\begin{aligned}
x[x := N] &= N \\
y[x := N] &= y; y \neq x \\
(E_1 E_2)[x := N] &= E_1[x := N] E_2[x := N] \\
(\lambda x. E)[x := N] &= \lambda x. E \\
(\lambda y. E)[x := N] \ (y \neq x) &= \lambda z. E[y := z][x := N], \\
&\text{where } y = z \text{ if } x \notin FV(E) \vee y \notin FV(N), \text{ otherwise } z \text{ is a fresh variable.} \\
(\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E)[x := N] \\
&= \text{letrec } z_1 = E_1[\vec{x}_i := \vec{z}_i][x := N], \dots, z_n = E_n[\vec{x}_i := \vec{z}_i][x := N] \\
&\text{in } E[\vec{x}_i := \vec{z}_i][x := N], \\
&\text{where } z_i = x_i \text{ if } x \notin FV(\text{letrec } x_1 = E_1, \dots, x_n = E_n \text{ in } E) \vee x_i \notin FV(N), \\
&\text{otherwise } z_i \text{ is a fresh variable } (i=1..n).
\end{aligned}$$

Definition 7 Given $x \in FV(E)$ and a context $C[]$, we say that x is free over $C[E]$ only if $\forall e, e \in \text{sub}(E, C) \Rightarrow x \in FV(e)$. 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 .

Lemma 2 Substitution Lemma: If $x \neq y$, and $x \notin FV(L)$, then

$$E[x := E'][y := L] = E[y := L][x := E'[y := L]]$$

Proof. Proof by induction on the structure of E .

Proposition 1 $E = E' \Rightarrow M[E] = M[E']$

Proof. Proof by induction on the structure of $M[]$.

4 FORMALISATION OF GENERALISE A DEFINITION

By *generalising a definition*, we mean to generalise a definition by making an identified expression of its right-hand side into a value passed into the function via a new formal parameter.

4.1 Definition of generalise a definition

Definition 8 Given an expression $\text{letrec } x_1 = E_1, \dots, x_i = E_i, \dots, x_n = E_n \text{ in } E_0$ 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:

$$x_i \notin FV(E') \wedge \forall x, e : (x \in FV(E') \wedge e \in \text{sub}(E_i, C) \Rightarrow x \in FV(e)).$$

After generalisation, the original expression becomes:

letrec $x_1 = E_1[x_i := x_i E']$, \dots , $x_i = \lambda z. (C[z][x_i := x_i z])$, \dots , $x_n = E_n[x_i := x_i E']$
in $E_0[x_i := x_i E']$, where z is a fresh variable

What follows is some explanation about the above definition:

- The condition $x_i \notin FV(E')$ 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 $x_i = E_i$.

4.2 Behaviour-preservation of *generalise 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.

Proof. Given the original expression:

$$\text{letrec } x_1 = E_1, \dots, x_i = E_i, \dots, x_n = E_n \text{ in } E$$

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

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

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

This does not change the semantics as x'_i is not used. Formally, the equivalence of semantics is guaranteed by rule 8 and the commutability of bindings within **letrec**.

Step 2. By the side-conditions and lemma 1, we can prove $x'_i E' = (\lambda z. C[z])E'_i = C[z][z := E'] = C[E'] = x_i$. Therefore substitute $x'_i E'_i$ for the free occurrences of x_i in the right-hand-side of x'_i does not change the semantics of x'_i , and we get

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

Step 3. In the definition of x'_i , replace E' with z , and extend the scope of λz to include the substitution $[x_i := x'_i z]$. we get

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

It is trivial to prove that the x'_i defined in this step is not semantically equal to the x'_i defined in step 2. However we can prove the equivalent of $x'_i E'$ from step 2 to step 3 in the context of the bindings for x_1, \dots, x_n . Let's use $x'_{i(E')}$ and $x'_{i(z)}$ to represent the x'_i s defined in step 2 and 3 respectively. Then

$$\begin{aligned}
x'_{i(z)} E' &= (\lambda z. (C[z][x_i := x'_{i(z)} z])) E' \\
&= C[z][x_i := x'_{i(z)} z][z := E'] \quad \text{by rule 1} \\
&= C[z][z := E'][x_i := x'_{i(z)} z][z := E'] \quad \text{by the substitution lemma} \\
&= C[E'][x_i := x'_{i(z)} E'] \quad \text{by lemma 1} \\
&= E_i[x_i := x'_{i(z)} E']
\end{aligned}$$

In a similar way, we can derive: $x'_{i(E')} E' = E_i[x_i := x'_{i(E')} E']$. The equivalent of $x'_{i(E')} E'$ between $x'_{i(z)} E'$ can be proved coinductively. As $x_i = x'_{i(E')} E'$, we have $x_i = x'_{i(z)} E'$.

Step 4. Substitute $x'_i E'$ for the occurrences of x_i outside the definition of x_i and x'_i does not change the semantics of the let-expression, as $x_i = x'_{i(z)} E'$ from step 3.

$$\begin{aligned}
&\text{letrec } x_1 = E_1[x_i := x'_i E'], \dots, x_i = E_i, \\
&\quad x'_i = \lambda z. (C[z][x_i := x'_i z]), \dots, x_n = E_n[x_i := x'_i E'] \\
&\text{in } E[x_i := x'_i E']
\end{aligned}$$

Step 5. Remove the definition of x_i , we get

$$\begin{aligned}
&\text{letrec } x_1 = E_1[x_i := x'_i E'], \dots, x'_i = \lambda z. C[z][x_i := x'_i z], \dots, x_n = E_n[x_i := x'_i E'] \\
&\text{in } E[x_i := x'_i E']
\end{aligned}$$

This does not change the semantics because rule 8 and the commutability of bindings within **letrec**.

Step 6. Rename x'_i to x_i , we have

$$\begin{aligned}
&\text{letrec } x_1 = E_1[x_i := x'_i E'][x'_i := x_i], \dots, \\
&\quad x_i = \lambda z. C[z][x_i := x'_i z][x'_i := x_i], \dots, x_n = E_n[x_i := x'_i E'][x'_i := x_i] \\
&\text{in } E[x_i := x'_i E'][x'_i := x_i]
\end{aligned}$$

Renaming bound variables, i.e. α -renaming, does not change the semantics. Finally, by the substitution lemma, we have

$$\begin{aligned}
&\text{letrec } x_1 = E_1[x_i := x_i E'], \dots, x_i = \lambda z. C[z][x_i := x_i z], \dots, x_n = E_n[x_i := x_i E'] \\
&\text{in } E[x_i := x_i E']
\end{aligned}$$

According to the definition of generalising a definition, this refactoring could introduce duplicated computation. One way to avoid duplicating the computation of $x_i E'$ is to introduce a new binding to represent the expression, instead of duplicating it at each call-site of x_i .

5 FORMALISATION OF A SIMPLE MODULE SYSTEM λ_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. A typical module-aware refactoring is *move a definition from one module to another*. This refactoring moves an identified declaration from its current module to a specified target 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. In order to formalise module-aware refactorings, we extend λ_{Letrec} with a simple module system. The definition of the new language, which is called λ_M , is given in the next section.

5.1 The Simple Module System λ_M

The syntax of λ_M terms is defined as:

$$\begin{aligned}
 \textit{Program} &::= \textit{let } \textit{Mod} \textit{ in } (\textit{Exp}; \textit{Imp}; \textit{letrec } D \textit{ in } E) \\
 \textit{Mod} &::= \varepsilon \mid \textit{Modid} = (\textit{Exp}; \textit{Imp}; D) \mid \textit{Mod}; \textit{Mod} \\
 \textit{Exp} &::= \varepsilon \mid (\textit{Exp}'_1, \dots, \textit{Exp}'_n) \quad (n \geq 0) \\
 \textit{Exp}' &= x \mid \textit{Modid}.x \mid \textit{module } \textit{Modid} \\
 \textit{Imp} &::= (\textit{Imp}'_1, \dots, \textit{Imp}'_n) \quad (n \geq 0) \\
 \textit{Imp}' &= \textit{import } \textit{Qual } \textit{Modid } \textit{Alias } \textit{ImpSpec} \\
 \textit{Modid} &::= M_i \quad (i \geq 0) \\
 \textit{Qual} &::= \varepsilon \mid \textit{qualified} \\
 \textit{ImpSpec} &::= \varepsilon \mid (x_1, \dots, x_n) \mid \textit{hiding } (x_1, \dots, x_n) \quad (n \geq 0) \\
 \textit{Alias} &::= \varepsilon \mid \textit{as } \textit{Modid} \\
 V &::= x \mid \textit{Modid}.x \mid \lambda x. E \\
 E &::= V \mid E_1 E_2 \mid \textit{letrec } D \textit{ in } E \\
 D &::= \varepsilon \mid x = E \mid D, D
 \end{aligned}$$

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 $(\textit{Exp}; \textit{Imp}; \textit{letrec } D \textit{ in } E)$ part in the definition of *Program* represents the *Main* module of the program, and the expression *E* represents the *main* expression. ε 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 [11].

The module system has been defined to model the Haskell 98 module system. Because only value variables can be defined in λ_M , λ_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 [7], 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. Although the term “relation” is used in the specification, a name should only refer to one entity in a valid Haskell program.

In the following specification of module-aware refactorings, we assume that, by using the module system analysis algorithm from the formal specification given in [7], 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, 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.

6 FUNDAMENTALS OF λ_M

Definition 9 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 10 Given a module $M=(Exp, Imp, D)$, we say module M is exported by itself if Exp is ϵ or module M occurs in Exp as an element of the export list.

Definition 11 The defining module of an identifier is the name of the module in which the identifier is defined.

Definition 12 Suppose v is an identifier that is in scope in module M , we use $defineMod(v, M)$ to represent the name of the module in which the identifier is defined.

Definition 13 $TBV(D)$ is the set of top-level identifiers declared in D (a sequence of declarations) and can be defined as:

$$\begin{aligned} TBV(\varepsilon) &= \{ \} \\ TBV(x = E) &= \{ x \} \\ TBV(D_1, D_2) &= TBV(D_1) \cup TBV(D_2) \end{aligned}$$

Definition 14 $FV(D)$ is the set of free variables in D (a sequence of declarations), and can be defined as:

$$\begin{aligned} FV(\varepsilon) &= \{ \} \\ FV(x = E) &= FV(E) - \{ x \} \\ FV(D_1, D_2) &= FV(D_1) \cup FV(D_2) - TBV(D_1, D_2) \end{aligned}$$

Definition 15 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: $B \subset \cup (D_i \times (D_i \cup I_i))_{i=1..n}$, and $\{(m_1, id_1), (m_2, id_2)\} \in B \mid id_2$ occurs free in the definition of id_1 ; id_1 's defining module is m_1 , and id_2 's defining module is m_2 }.

Definition 16 We say that the identifier x defined in module N is used by module $M = (Exp, Imp, D)$ ($M \neq N$) if $DefineMod(x, M) = N$ and either $x \in FV(D)$ or x is exported by module M , otherwise we say that the x defined in module N is not used by module M .

Definition 17 Given a set of identifiers Y and an export list Exp , $rmFromExp(Exp, Y)$ removes the occurrences of the identifiers of Y from Exp , and is defined as:

$$\begin{aligned} rmFromExp(\varepsilon, Y) &= \varepsilon \\ rmFromExp((), Y) &= () \\ rmFromExp((e, Exp'_2, \dots, Exp'_n), Y) \quad (e \notin Y) \\ &= (e, rmFromExp(Exp'_2, \dots, Exp'_n), Y) \\ rmFromExp((e, Exp'_2, \dots, Exp'_n), Y) \quad (e \in Y) \\ &= rmFromExp((Exp'_2, \dots, Exp'_n), Y) \end{aligned}$$

Definition 18 Given an identifier y which is defined in module M , and the export list, Exp , of module M , $addToExp(Exp, y, M)$ ensures that module M exports y , and can be defined as:

$$\begin{aligned} addToExp(\varepsilon, y, M) &= \varepsilon \\ addToExp((), y, M) &= (y) \\ addToExp((Exp'_1, \dots, Exp'_n), y, M) \\ &= (Exp'_1, \dots, Exp'_n) \text{ if } \exists i, y \equiv Exp'_i \quad (1 \leq i \leq n); \\ &\text{otherwise } (Exp'_1, \dots, Exp'_n, y) \end{aligned}$$

Similar to the above two definitions, the following three 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 19 Given an identifier y which is exported by module M and Imp which is a sequence of imports, $rmFromImp(Imp, y, M)$ removes the literal occurrences of y in the import declarations that import M . The function can be used to cleanup the uses of y in import declarations that import module M when y is no longer exported by M .

Definition 20 Given an identifier y which is exported by module M (M is not necessarily the module where y is defined) and Imp which is a sequence of imports, then $hideInImp(Imp, y, M)$ ensures that Imp does not bring this identifier into scope by importing it from module M .

Definition 21 Suppose the same binding, say y , is exported by both module M_1 and M_2 , and Imp is a sequence of import declarations, then $chgImpPath(Imp, y, M_1, M_2)$ switches the importing of y from M_1 to M_2 .

7 FORMALISATION OF MOVE A DEFINITION FROM ONE MODULE TO ANOTHER IN λ_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 $f_{\circ\circ}$ from module M to module N , here are some design decisions we have made during the implementation of this refactoring in HaRe.

- If a variable which is free in the definition of $f_{\circ\circ}$ 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 $f_{\circ\circ}$ is already in scope in module N (either defined by module N or imported from other modules), but it refers to another $f_{\circ\circ}$ other than the one defined in module M , the user will be prompted to do renaming first.
- Mutually recursive modules should not be introduced by the refactoring. Although mutually recursive modules are allowed in Haskell 98, transparent compilation of these mutually recursive modules are not yet supported by the current working Haskell compilers/interpreters. Therefore, we try to avoid introducing mutually recursive modules during refactoring.
- If $f_{\circ\circ}$ is exported by module M before the refactoring, then module M still exports $f_{\circ\circ}$ after the refactoring as long as doing this does not introduce recursive modules; If $f_{\circ\circ}$ is not exported by module M , then module M does not export $f_{\circ\circ}$ after the refactoring.
- Module N will export $f_{\circ\circ}$ after the refactoring only if $f_{\circ\circ}$ is either exported by module M or used by the other definitions in module M before the refactoring. The imports of $f_{\circ\circ}$ will be via M if module M still exports $f_{\circ\circ}$ after the refactoring; otherwise via N .

7.1 Definition of move a definition from one module to another

The following definition defines *move a definition from one module to another*. A commentary on the definition follows.

Definition 22 Given a valid program P :

$$P = \text{let } M_1 = (\text{Exp}_1; \text{Imp}_1; x_1 = E_1, \dots, x_i = E_i, \dots, x_n = E_n); \\ M_2 = (\text{Exp}_2; \text{Imp}_2; D_2); \dots; M_m = (\text{Exp}_m; \text{Imp}_m; D_m) \\ \text{in } (\text{Exp}_0; \text{Imp}_0; \text{letrec } D_0 \text{ in } E)$$

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

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

To make the specification clear, the program after the refactoring, P' , is given by two cases according to whether x_i is exported by M_1 . In each case, different situations are considered.

Case 1. x_i is not exported by M_1 .

Case 1.1. x_i is not used by other definitions in M_1 , i.e.

$$\{x_i, M_1.x_i\} \cap FV(E_{j(j \neq i)}) = \emptyset$$

$$P' = \text{let } M_1 = (\text{Exp}_1; \text{Imp}_1; x_1 = E_1, \dots, x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, \dots, x_n = E_n); \\ M_2 = (\text{Exp}_2; \text{Imp}_2; x_i = E_i[M_1.x_i := M_2.x_i], D_2); \dots; \\ M_m = (\text{Exp}_m; \text{Imp}'_m; D_m) \\ \text{in } (\text{Exp}_0; \text{Imp}'_0; \text{letrec } D_0 \text{ in } E)$$

where

$$\text{Imp}'_j = \text{hideInImp}(\text{Imp}_j, x_i, M_2) \text{ if } M_2 \text{ is exported by itself; } \text{Imp}_j \text{ otherwise. } (3 \leq j \leq m)$$

Case 1.2. x_i is used by other definitions in M_1 .

$$P' = \text{let } M_1 = (\text{Exp}_1; \text{Imp}'_1; x_1 = E_1, \dots, x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, \dots, x_n = E_n); \\ M_2 = (\text{Exp}'_2; \text{Imp}_2; x_i = E_i[M_1.x_i := M_2.x_i], D_2); \dots; \\ M_m = (\text{Exp}_m; \text{Imp}'_m; D_m) \\ \text{in } (\text{Exp}_0; \text{Imp}'_0; \text{letrec } D_0 \text{ in } E)$$

where

$$\text{Imp}'_1 = \text{hideInImp}(\text{Imp}_1, x_i, M_2); \text{import } M_2 \text{ as } M_1(x_i) \\ \text{Exp}'_2 = \text{addToExp}(\text{Exp}_2, x_i, M_2) \\ \text{Imp}'_j = \text{hideInImp}(\text{Imp}_j, x_i, M_2) \quad (3 \leq j \leq m)$$

Case 2. x_i is exported by M_1 .

Case 2.1. M_2 is not a client module of M_1 .

$$\begin{aligned}
P' = \text{let } & M_1 = (\text{Exp}_1; \text{Imp}'_1; x_1 = E_1, \dots, x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, \dots, x_n = E_n); \\
& M_2 = (\text{Exp}'_2; \text{Imp}_2; x_i = E_i[M_1.x_i := M_2.x_i], D_2); \dots; \\
& M_m = (\text{Exp}_m; \text{Imp}'_m; D_m) \\
& \text{in } (\text{Exp}_0; \text{Imp}'_0; \text{letrec } D_0 \text{ in } E)
\end{aligned}$$

where

$$\begin{aligned}
\text{Imp}'_1 &= \text{Imp}_1; \text{import } M_2 \text{ as } M_1(x_i) \\
\text{Exp}'_2 &= \text{addToExp}(\text{Exp}_2, x_i, M_2) \\
\text{Imp}'_j &= \text{hideInImp}(\text{Imp}_j, x_i, M_2) \quad (3 \leq j \leq m)
\end{aligned}$$

Case 2.2. M_2 is a client module of M_1 .

$$\begin{aligned}
P' = \text{let } & M_1 = (\text{Exp}'_1; \text{Imp}_1; x_1 = E_1, \dots, x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, \dots, x_n = E_n); \\
& M_2 = (\text{Exp}'_2; \text{Imp}'_2; x_i = E_i[M_1.x_i := M_2.x_i], D_2); \dots; \\
& M_m = (\text{Exp}_m; \text{Imp}'_m; D_m) \\
& \text{in } (\text{Exp}_0; \text{Imp}'_0; \text{letrec } D_0 \text{ in } E)
\end{aligned}$$

where

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

What follows is some explanation about the above definition:

- As to the side-conditions, condition a) 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. This condition aims to avoid causing conflict/ambiguous occurrence [11] in M_2 ; condition b) requires that all the free variables used in the definition of x_i are in scope in M_2 . An entity can be brought into scope by either refactoring the exports/imports of the involved modules, or using the *move a definition from one module to another* refactoring to move the definition into scope. While it is possible for this refactoring itself to bring those variables into scope, doing this will make its definition and implementation more complicated, therefore we chose to divide the functionality into more elementary refactorings; finally, conditions c) and d) guarantee that mutual recursive modules won't be introduced during the refactoring process.

- The design of transformation rules was made complicated mainly by two reasons.

The first reason is that it is not clear whether M_1 should still export x_i after the refactoring if it does before the refactoring. After having examined a number of examples with the original module, M_1 , and the target module, M_2 , having different relationships in the module graph, we concluded that the answer to whether M_1 should still export x_i depends on the concrete situation and the user's intention. In this specification, we choose to let M_1 still export x_i whenever possible.

The second reason is due to the Haskell 98 module system. The module system of Haskell 98 is simple and flexible, but not very powerful at controlling the export list. For example, in Haskell 98, an entity in the export list can be of the form “Module M”, which represents the set of all entities that are in scope with both an unqualified name “e” and a qualified name “M.e”. But unlike the case for import declarations, where entities can be excluded by using `hiding (imp1, ..., impn)`, there is no such mechanism with exports. Therefore, when “module M” is used in the export list, no entity which is in scope with both an unqualified name “e” and a qualified name “M.e” can be excluded from being exported. Another example is that if the export list in a Haskell 98 module is omitted, then all values, types and classes defined in the module are exported. The only way to exclude some entities from being exported is to use an explicit list to specify those exported entities. This is inconvenient however when the developer wants to expose most of, but not all of the defined entities in the module.

One of the inconveniences caused by this lack of control in the export list is: 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. While it is possible to check each potentially affected module to detect these problems, it will certainly slow down the refactoring process.

Two strategies are used in the transformation in order to overcome the inconvenience caused by this lack of control in the export list. 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 shortcoming of this strategy is that it involves the client modules of the current module; The second strategy is to use *alias* in the import declaration to avoid the changes to the export list. This is used in the specification of Imp'_1 in case 1.2, where `import M2 as M1(xi)` is used to ensure that the interface of module M_1 stays unchanged.

7.2 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:

$$B' = \{(fx, fy) \mid (x, y) \in B\},$$

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.

8 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 λ_{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 refactorings. 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 [16], 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

- [1] The Emacs Editor. <http://www.gnu.org/software/emacs/>.
- [2] The Vim Editor. <http://www.vim.org/>.
- [3] The XEmacs Editor. <http://www.xemacs.org/>.
- [4] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 1995.
- [5] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [6] J. Darlington. Program Transformations. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.
- [7] I. S. Diatchki, M. P. Jones, and T. Hallgren. A Formal Specification for the Haskell 98 Module System. In *ACM Sigplan Haskell Workshop*, 2002.
- [8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [9] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
- [10] S. P. Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP*, pages 18–44, 1996.
- [11] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999.
- [12] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, August 2003.
- [13] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer: HaRe, and its API. In John Boyland and Grel Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005.
- [14] H. Parnsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3), September 1983.
- [15] G. D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [16] Refactor-fp. Refactoring Functional Programs. <http://www.cs.kent.ac.uk/projects/refactor-fp/>.